Certified Tester Advanced Level Test Automation Engineering Syllabus

Version 2.0

International Software Testing Qualifications Board

Copyright Notice

Revision History

Version	Date	Remarks
Syllabus 2016	2016/10/21	CTAL-TAE GA Release
Syllabus v2.0	2024/05/03	CTAL-TAE v2.0 GA Release

Table of Contents

Acknowledgements

This document was formally released by the General Assembly of the ISTQB® on May 3, 2024.

It was produced by the Test Automation Task Force of the Specialist Working Group from the International Software Testing Qualifications Board: Graham Bath (Specialist Working Group Chair) Andrew Pollner (Specialist Working Group Vice Chair and Test Automation Task Force Chair), Péter Földházi, Patrick Quilter, Gergely Ágnecz, László Szikszai. Test Automation Task Force reviewers included: Armin Beer, Armin Born, Geza Bujdoso, Renzo Cerquozzi, Jan Giesen, Arnika Hryszko, Kari Kakkonen, Gary Mogyorodi, Chris van Bael, Carsten Weise, Marc-Florian Wendland.

Technical Reviewer: Gary Mogyorodi

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

Horváth Ágota, Laura Albert, Remigiusz Bednarczyk, Jürgen Beniermann, Armin Born, Alessandro Collino, Nicola De Rosa, Wim Decoutere, Ding Guofu, Istvan Forgacs, Elizabeta Fourneret, Sudhish Garg, Jan Giesen , Matthew Gregg, Tobias Horn, Mattijs Kemmink, Hardik Kori, Jayakrishnan Krishnankutty, Ashish Kulkarni, Vincenzo Marrazzo, Marton Matyas, Patricia McQuaid, Rajeev Menon, Ingvar Nordström, Arnd Pehl, Michaël Pilaeten, Daniel Polan, Nishan Portoyan, Meile Posthuma, Adam Roman, Pavel Sharikov, Péter Sótér , Lucjan Stapp, Richard Taylor, Giancarlo Tomasig, Chris Van Bael, Koen Van Belle, Johan Van Berkel, Carsten Weise, Marc-Florian Wendland, Ester Zabar.

ISTQB Working Group Advanced Level Test Automation Engineer (Edition 2016): Andrew Pollner (Chair), Bryan Bakker, Armin Born, Mark Fewster, Jani Haukinen, Raluca Popescu, Ina Schieferdecker.

0 Introduction

0.1 Purpose of this Syllabus

This syllabus forms the basis for the International Software Testing Qualification for the Advanced Level Test Automation Engineering (CTAL-TAE) qualification. The ISTQB® provides this syllabus as follows:

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

0.2 Test Automation Engineering in Software Testing

The Test Automation Engineering qualification is aimed at anyone involved in software testing and test automation. This includes people in roles such as testers, test analysts, test automation engineers, test consultants, test architects, test managers, and software developers. This qualification is also appropriate for anyone who wants a basic understanding of test automation, such as project managers, quality managers, software development managers, business analysts, IT directors and management consultants.

The Test Automation Engineering syllabus is targeted to the test engineer looking to implement or improve on test automation. It defines methods and practices that can support a sustainable solution.

Other guidelines and reference models relating to test automation solutions are software engineering standards for the selected software development lifecycles, programming technologies, and formatting standards. This syllabus does not teach software engineering. However, a test automation engineer is expected to have skills, experience, and expertise in software engineering.

Furthermore, a test automation engineer needs to be aware of industry programming and documentation standards and best practices to make use of them while developing a test automation solution. These practices can increase maintainability, reliability, and security of the test automation solution. Such standards are typically based on quality characteristics.

0.3 Career Path for Testers and Test Automation Engineers

The ISTQB® scheme provides support for testing professionals at all stages of their careers offering both breadth and depth of knowledge. Individuals who achieve the ISTQB® Test Automation Engineering certification may also be interested in the Test Automation Strategy (CT-TAS) qualification.

Individuals who achieve the ISTQB® Certified Tester Test Automation Engineering certification may also be interested in the Core Advanced Levels (Test Analyst, Technical Test Analyst, and Test Manager) and thereafter Expert Level (Test Management or Improving the Test Process). Anyone seeking to develop skills in testing practices in an Agile environment area could consider the Agile Technical Tester or Agile Test Leadership at Scale certifications. The Specialist stream offers a deep dive into areas that have specific test approaches and test activities e.g., in Test Automation Strategy, Performance Testing, Security Testing, AI Testing, and Mobile Application Testing, or where domain specific knowledge is required (e.g., Automotive Software Testing or Game Testing). Please visit www.istqb.org for the latest information of ISTQB´s Certified Tester Scheme.

0.4 Business Outcomes

This section lists the Business Outcomes expected of a candidate who has achieved the Test Automation Engineering certification.

A candidate who has achieved the Test Automation Engineering certification can…

TAE-B01	Describe the purpose of test automation
TAE-B02	Understand test automation through the software development lifecycle
TAE-B03	Understand the Configuration of an Infrastructure to Enable Test Automation
TAE-B04	Learn the evaluation process for selecting the right tools and strategies
TAE-B05	Understand design concepts for building modular and scalable test automation solutions
TAE-B06	Select an approach, including a pilot, to plan test automation deployment within the software development lifecycle
TAE-B07	Design and develop (new or modified) test automation solutions that meet technical needs
TAE-B08	Consider scope and approach of test automation and maintenance of testware
TAE-B09	Understand how automated tests integrate within CI/CD pipelines
TAE-B10	Understand how to collect, analyze, and report on test automation data in order to inform stakeholders
TAE-B11	Verify the test automation infrastructure
TAE-B12	Define continuous improvement opportunities for test automation

0.5 Examinable Learning Objectives and Cognitive Level of Knowledge

Learning objectives support business outcomes and are used to create the Certified Tester Test Automation Engineering exams.

In general, all contents of this syllabus are examinable at a K2, K3 and K4 levels, except for the Introduction and Appendices. That is, the candidate may be asked to recognize, remember, or recall a keyword or concept mentioned in any of the eight chapters. The specific learning objectives levels are shown at the beginning of each chapter, and classified as follows:

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

Further details and examples of learning objectives are given in Appendix A.

All terms listed as keywords just below chapter headings shall be remembered, even if not explicitly mentioned in the learning objectives.

0.6 The Test Automation Engineering Certificate Exam

The Test Automation Engineering Certificate exam will be based on this syllabus. Answers to exam questions may require the use of material based on more than one section of this syllabus. All sections of the syllabus are examinable, except for the Introduction and Appendices. Standards and books are included as references, but their content is not examinable, beyond what is summarized in the syllabus itself from such standards and books.

Refer to the Exam Structures and Rules v1.1 Compatible with Syllabus Foundation and Advanced Levels and Specialists Modules document for further details regarding the Test Automation Engineering Certificate exam.

The entry criterion for taking the Test Automation Engineering exam is that candidates have an interest in software testing and test automation. However, it is strongly recommended that candidates also:

• Have at least a minimal background in software development and software testing, such as six months experience as a software test engineer or as a software developer.

• Take a course that has been accredited to ISTQB standards (by one of the ISTQB-recognized member boards).

Entry Requirement Note: The ISTQB® Foundation Level certificate shall be obtained before taking the ISTQB® Test Automation Engineering certification exam.

0.7 Accreditation

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

The accreditation guidelines for this syllabus follow the general Accreditation Guidelines published by the Processes Management and Compliance Working Group.

0.8 Handling of Standards

There are standards referenced in the Test Automation Engineering syllabus (e.g., (IEEE, and ISO). The purpose of these references is to provide a framework (as in the references to ISO 25010 regarding quality characteristics) or to provide a source of additional information if desired by the reader. Please note that the syllabus uses the standard documents as reference. The standards documents are not intended for examination. Refer to Chapter 9 References for more information on Standards.

0.9 Keeping It Current

The software industry changes rapidly. To deal with these changes and to provide the stakeholders with access to relevant and current information, the ISTQB working groups have created links on the www.istqb.org website, which refer to supporting documents and changes to standards. This information is not examinable under the Test Automation Engineering syllabus.

0.10 Level of Detail

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

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

The syllabus content is not a description of the entire knowledge area of software testing; it reflects the level of detail to be covered in Test Automation Engineering training courses. It focuses on test concepts and techniques that can apply to all software projects, including those following Agile methods. This syllabus does not contain any specific learning objectives related to Agile testing, but it does discuss how these concepts apply in Agile projects and other types of projects.

0.11 How this Syllabus is Organized

There are eight chapters with examinable content. The top-level heading for each chapter specifies the time for the chapter; timing is not provided below chapter level. For accredited training courses, the syllabus requires a minimum of 21 hours of instruction, distributed across the eight chapters as follows:

- Chapter 1: 45 minutes – Introduction and Objectives for Test Automation
  - The tester learns about the benefits of test automation and its limitations
  - Test automation within different software development lifecycle models is covered
  - The tester learns how a system under test (SUT) architecture impacts the suitability of test tools
- Chapter 2: 180 minutes – Preparing for Test Automation
  - Design for SUT testability through observability, controllability, and a clearly defined architecture, are covered.
  - A tester learns about test automation across different environments
  - Factors required to assess an appropriate test automation solution are covered
  - A tester will learn about the technical considerations needed to develop recommendations on test automation
- Chapter 3: 210 minutes – Test Automation Architecture
  - Test automation architecture and its components leading to a test automation solution is covered.
  - A tester will learn about layers and their application in a test automation framework
  - Multiple approaches to using test automation tools will be covered
  - A tester will learn how design principles and design patterns can be applied to test automation
- Chapter 4: 150 minutes – Implementing Test Automation
  - How to effectively plan and deploy a test automation pilot project will be covered
  - A tester will learn about deployment risks and mitigate strategies
  - Factors that improve maintainability of test automation code will be covered
- Chapter 5: 90 minutes – Implementation and Deployment Strategies for Test Automation
  - A tester will learn about CI/CD pipelines and automated test execution across test levels
  - Configuration management for components of test automation will be covered
  - A tester will learn about dependencies applied to API and contract testing

- Chapter 6: 150 minutes – Test automation Reporting and Metrics
    - A tester will learn about where data can be collected from an SUT and test automation for analysis and reporting
    - Data analysis from SUT reports and test automation to uncover causes of failures will be covered
    - Use of test reports and dashboards to inform stakeholders will be covered
- Chapter 7: 135 minutes – Verifying the Test Automation Solution
    - The tester will learn how to examine and verify the correct operation of test automation components and environment
    - Ensuring that test scripts and test suites execute correctly will be covered
    - A tester will understand when to perform root cause analysis
    - Techniques to analyze test automation code for quality will be covered
- Chapter 8: 210 minutes – Continuous Improvement
    - Additional areas of data analysis for test case improvement will be covered
    - A tester will learn ways to make improvements and upgrades to a test automation solution and its components
    - Identifying ways to consolidate and streamline test automation will be covered
    - A tester will learn how test automation tools can assist with test support and setup needs

1 Introduction and Objectives for Test Automation – 45 minutes (K-2)

Keywords

system under test, test automation, test automation engineer

Learning Objectives for Chapter 1:

1.1 Purpose of Test Automation

TAE-1.1.1 (K2) Explain the advantages and disadvantages of test automation

1.2 Test Automation in the Software Development Lifecycle

TAE-1.2.1 (K2) Explain how test automation is applied across different software development lifecycle models

TAE-1.2.2 (K2) Select suitable test automation tools for a given system under test

1.1 Purpose of Test Automation

1.1.1 Explain the Advantages and Disadvantages of Test Automation

Test automation, which includes automated test execution and test reporting, is one or more of the following activities:

- Using purpose-built software tools to control and set up test suites for test execution
- Executing tests in an automated way
- Comparing actual results to expected results

Test automation provides significant features and capabilities that can interact with a system under test (SUT). Test automation can cover a broad area of software. Solutions cover many kinds of software (e.g., SUT with a user interface (UI), SUT without a UI, mobile applications, network protocols, and connections).

Test automation has many advantages. It:

- Allows more tests to be run per build compared to manual tests
- Provides the ability to create and execute tests that cannot be executed manually (e.g., real time response, remote testing, and parallel testing)
- Allows for tests that are more complex than manual tests
- Executes faster than manual tests
- Is less subject to human error
- Is more effective and efficient in use of test resources
- Provides quicker feedback regarding SUT quality
- Helps improve system reliability (e.g., availability, and recoverability)
- Improves the consistency of test execution across test cycles

However, test automation has potential disadvantages including:

- Additional costs will be involved for the project as there may be a need to hire a test automation engineer (TAE), buy new hardware, and set up training
- Requirement of initial investment to set up a test automation solution
- Time to develop and maintain a test automation solution
- Requirement for clear test automation objectives to ensure success
- Rigidity of tests, and less adaptability to changes in the SUT
- Introduction of additional defects by test automation

There are limitations to test automation that need to be kept in mind:

- Not all manual tests can be automated
- Verifying only what automated tests are programmed to do
- Test automation can only check machine interpretable test results which means that some quality characteristics may not be testable with automation
- Test automation can only check test results that can be verified by an automated test oracle

1.2 Test Automation in the Software Development Lifecycle

1.2.1 Explain How Test Automation is Applied Across Different Software Development Lifecycle Models

Waterfall

The waterfall model is an SDLC model that is both linear and sequential. This model has distinct phases (i.e., requirements, design, implementation, verification, and maintenance) and each phase typically concludes with documentation that must be approved. Implementation of test automation typically happens in parallel to or after the implementation phase. Test runs usually take place during the verification phase due to the software components not being ready for testing until then.

V-model

The V-model is an SDLC model where a process is executed in a sequential manner. As a project is defined from high level requirements to low-level requirements, corresponding test and integration activities are defined to validate those requirements. This is where the traditional test levels are derived from: component, component integration, system, system integration, and acceptance as described in the CTFL Section 2.2. Providing a test automation framework (TAF) for each test level is possible and recommended.

Agile software development

In Agile software development, there are countless possibilities for test automation. Unlike the waterfall or the V-model, in the Agile software development method, TAEs and business representatives can decide on the roadmap, timeline and planned test delivery. In this method, there are best practices such as code reviews, pair programming and frequent automated test execution. Eliminating silos (i.e., making sure that developers, testers, and other stakeholders work together) allows teams to cover all test levels with the appropriate amount and depth of test automation, achieving a goal called in-sprint automation. More details are found in the ISTQB CT-TAS Syllabus, Section 3.2.

1.2.2 Select Suitable Test Automation Tools For a Given System Under Test

To identify the most suitable test tools for a given project the SUT must first be analyzed. TAEs need to identify the project requirements that can be used as the baseline for tool selection.

Since different test automation tool features are used for UI software and, for example, web services, it is important to understand what the project wants to achieve over time. There is no limit to the number of test automation tools and features that can be used or selected, but the costs must always be considered. Using a commercial off-the-shelf tool or implementing a custom solution based on open-source technology can be a complex process.

The next topic to be evaluated is the composition and experience of the team in test automation. In the case where the testers have little to no programming experience, using a low-code or no-code solution may be a viable choice.

For technical testers with programming knowledge, it can be helpful to select tools whose language matches that of the SUT. This provides advantages including the ability to work with developers on debugging test automation defects more efficiently and cross training of members between teams.

2 Preparing for Test Automation – 180 minutes (K4)

Keywords

API testing, GUI testing, testability

Learning Objectives for Chapter 2:

2.1 Understand the Configuration of an Infrastructure to Enable Test Automation

TAE-2.1.1 (K2) Describe the configuration needs of an infrastructure that enable implementation of test automation

TAE-2.1.2 (K2) Explain how test automation is leveraged within different environments

2.2 Evaluation Process for Selecting the Right Tools and Strategies

TAE-2.2.1 (K4) Analyze a system under test to determine the appropriate test automation solution

TAE-2.2.2 (K4) Illustrate the technical findings of a tool evaluation

2.1 Understand the Configuration of an Infrastructure to Enable Test Automation

2.1.1 Describe the Configuration Needs of an Infrastructure that Enable Implementation of Test Automation

Testability of the SUT (i.e., availability of software interfaces that support testing, e.g., to enable control and observability of the SUT) should be designed and implemented in parallel with the design and implementation of the other features of the SUT. This work is generally performed by a software architect as testability is a non-functional requirement of the system, often with the involvement of a TAE to identify the specific areas where improvements can be made.

For better testability of the SUT there are different solutions that can be utilized which have different configuration needs, for example:

- Accessibility identifiers
  - The different development frameworks can generate these identifiers automatically or the developers can set them manually
- System environment variables
  - Certain application parameters can be changed to enable easier testing through administration
- Deployment variables
  - Similar to system variables but can be set before starting deployment

Designing for testability of a SUT consists of the following aspects:

- Observability: The SUT needs to provide interfaces that give insight into the SUT. Test cases can then use these interfaces to determine whether the actual results equal the expected results.
- Controllability: The SUT needs to provide interfaces that can be used to perform actions on the SUT. This can be UI elements, function calls, communication elements (e.g., Transmission Control Protocol/Internet Protocol (TCP/IP), and Universal Serial Bus (USB) protocols), or electronic signals for physical or logical switches on the different environment variables.
- Architecture transparency: The documentation of an architecture needs to provide clear, understandable components and interfaces that give observability and controllability at all test levels and foster quality.

2.1.2 Explain How Test Automation is Leveraged within Different Environments

Different types of automated tests can be executed in different environments. These environments can differ between projects and methodologies, and most projects have one or more environments to utilize for testing. From a technical point of view these environments can be created from containers, virtualization software, and using other approaches.

A set of possible environments to consider include the following:

Local development environment

A local development environment is where software is initially created, and components are tested with automation to verify functional suitability. Several test types can occur in the local development environment including component testing, GUI testing, and application programming interface (API) testing. It is also important to note that by using an integrated development environment (IDE) on the given computer, white box testing can be performed to identify poor coding and quality problems as early as possible.

Build environment

Its main purpose is to build the software and execute tests that check the correctness of the resulting build in a DevOps ecosystem. This environment can be either a local development environment or a continuous integration/continuous delivery (CI/CD) agent where low-level tests (i.e., component tests, and component integration tests) and static analysis can be performed without actual deployment to any other environments.

Integration environment

After performing low-level testing and static analysis, the next stage is a system integration environment. Here, a release candidate of the SUT is present that is fully integrated with other systems that can be tested. In this environment a fully automated test suite, either UI tests or API tests, can be executed. In this environment there is no white-box testing, only black-box testing (i.e., system integration and/or acceptance testing). It is important to note that this is the first environment where monitoring should be present to see what happens in the background during the use of the SUT to enable efficient investigation of defects/failures.

Preproduction environment

A preproduction environment is used mostly to assess non-functional quality characteristics (e.g., performance efficiency). Although non-functional tests can be performed in any environment, there is extra focus on preproduction because it resembles production as closely as possible. Often, user acceptance testing can be performed by business stakeholders to verify the final product and it is possible to execute the existing automated test suite here as well, if necessary. This environment is also monitored.

Production/Operational environment

A production environment can be used to assess functional and non-functional quality characteristics in real time while users interact with a deployed system with monitoring and certain best practices that enable testing in production (e.g. canary release, blue/green deployment, and A/B testing).

2.2 Evaluation Process for Selecting the Right Tools and Strategies

2.2.1 Analyze a System Under Test to Determine the Appropriate Test Automation Solution

Each SUT can be different from one another, yet there are various factors and characteristics that can be analyzed to have a successful test automation solution (TAS). During the investigation of an SUT, TAEs need to gather requirements considering its scope and given capabilities. Different kinds of applications (e.g., web service, mobile, and web) need different kinds of test automation from a technical point of view. The investigation can be done - and is recommended - in collaboration with other stakeholders (e.g. manual testers, business stakeholders, and business analysts) to identify as many risks and their mitigations as possible to have a beneficial test automation solution for the future.

The requirements for a test automation approach and architecture should consider the following:

- Which test process activities should be automated, (e.g., test management, test design, test generation, and test execution)
- Which test levels should be supported
- Which test types should be supported
- Which test roles and skill sets should be supported
- Which software products, product lines and families should be supported, (e.g., to define the span and lifetime of the implemented TAS)
- Which kind of SUTs need to be compatible with the TAS
- Availability of test data and its quality
- Possible methods and ways to emulate unreachable cases (e.g., 3rd party applications involved)

2.2.2 Illustrate the Technical Findings of a Tool Evaluation

After the SUT is analyzed and the requirements have been collected from all stakeholders, there are likely test automation tools that meet these requirements that can be considered. There may not be a single tool that fits all identified requirements, and stakeholders should recognize this possibility.

It is helpful to capture the findings about the possible tools and reflect on the various direct and indirect requirements in a comparison table. The goal of the comparison table is to allow stakeholders to see the differences between the tools based on specific requirements. The comparison table lists the tools in the columns, and the requirements in the rows. The cells contain information about the properties of each tool regarding each requirement and about priorities.

In general, test automation tools should be assessed to determine if they fit the requirement identified in the previous section (2.2.1). Requirements to consider when evaluating and comparing the tools include:

- The language/technology of the tool and the IDE tools
- The ability to configure a tool, whether it supports different test environments, runs configurations, and uses dynamic or static setup values
- The ability to manage test data within the tool. Test data management might be integrated with a central repository for version control.
- For different test types, different test automation tools might need to be selected
- The ability to provide reporting capability. This is important to align with the test reporting requirements of the project.
- The ability to integrate with other tools used on a project or in the organization, such as CI/CD, task tracking, test management, reporting or other tools
- The ability to expand overall test architecture and assess the scalability, maintainability, modifiability, compatibility, and reliability of tools

This comparison table is a good source to determine a proposed tool or tool set to use for test automation of the SUT.

The process can vary as to how the decision is made on which tool(s) will be used, but the proposal should be demonstrated to the appropriate stakeholders for approval.

3 Test Automation Architecture – 210 minutes (K3)

Keywords

behavior-driven development, capture/playback, data-driven testing, generic test automation architecture, keyword-driven testing, linear scripting, model-based testing, structured scripting, test adaptation layer, test automation framework, test automation solution, test harness, test script, testware, test step, test-driven development

Learning Objectives for Chapter 3:

3.1 Design Concepts Leveraged in Test Automation

TAE-3.1.1 (K2) Explain the major capabilities in a test automation architecture

TAE-3.1.2 (K2) Explain how to design a test automation solution

TAE-3.1.3 (K3) Apply layering of test automation frameworks

TAE-3.1.4 (K3) Apply different approaches for automating test cases

TAE-3.1.5 (K3) Apply design principles and design patterns in test automation

3.1 Design Concepts Leveraged in Test Automation

3.1.1 Explain the Major Capabilities in a Test Automation Architecture

Generic Test Automation Architecture (gTAA)

The gTAA is a high-level design concept that provides an abstract view of the communication between test automation and the systems that test automation is connected to, i.e., the SUT, project management, test management, and configuration management (see figure 1). It also provides the capabilities that are necessary to cover when designing a test automation architecture (TAA).

The interfaces of gTAA describe the following:

- The SUT interface describes the connection between the SUT and the TAF (see section 3.1.3 regarding the test automation framework).
- The project management interface describes the test automation development progress.
- The test management interface describes the mapping of test case definitions and automated test cases.
- Configuration management interface describes the CI/CD pipelines, environments and testware.

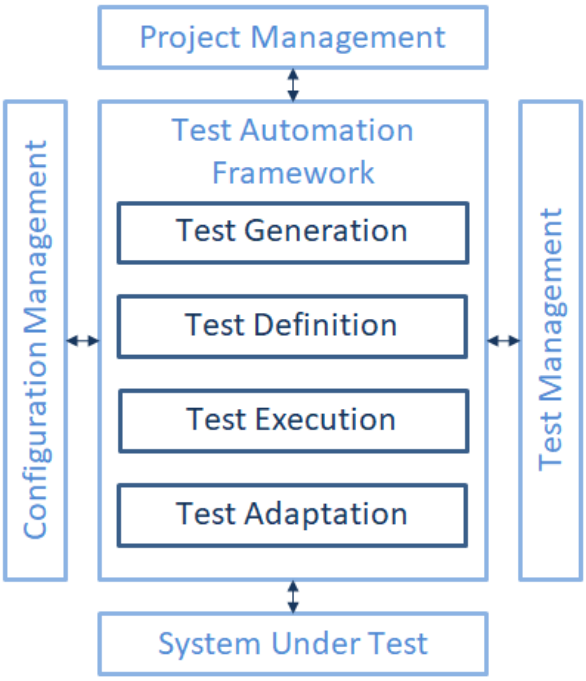

Figure 1: gTAA diagram

Capabilities provided by test automation tools and libraries

Core test automation capabilities should be identified and selected from available tools as required for a given project.

Test generation: Supports the automated design of test cases based on a test model. Model-based testing tools can be leveraged in the generation process (see the ISTQB CT-MBT Syllabus). Test generation is an optional capability.

Test definition: Supports the definition and implementation of test cases and/or test suites, which optionally can be derived from a test model. It separates the test definition from the SUT and/or test tools. It contains the means to define high-level and low-level tests, which are handled in the test data, test cases, and test library components or combinations thereof.

Test execution: Supports test execution and test logging. It provides a test execution tool to run the selected tests automatically, and a test logging and test reporting component.

Test adaptation: Provides the necessary functionality to adapt the automated tests for the various components or interfaces of the SUT. It provides different adaptors for connecting to the SUT via APIs, protocols, and services.

3.1.2 Explain How to Design a Test Automation Solution

A Test Automation Solution (TAS) is defined by an understanding of functional, non-functional, and technical requirements of the SUT, existing or required tools that are necessary to implement a solution. A TAS is implemented with commercial or open-source tools and may need additional SUT-specific adaptors.

The TAA defines the technical design for the overall TAS. It should address:

- Selecting test automation tools and tool specific libraries
- Developing plugins and/or components
- Identifying connectivity and interface requirements (e.g., firewalls, database, uniform resource locators (URLs)/connections, mocks/stubs, message queues, and protocols)
- Connecting to the test management and defect management tools
- Utilizing a version control system and repositories

3.1.3 Apply Layering of Test Automation Frameworks

Test Automation Framework

The TAF is the foundation of a TAS. It often includes a test harness, also known as test runner, and test libraries, test scripts and test suites.

TAF Layers

TAF layers define a distinct border of classes that have similar purposes such as test cases, test reporting, test logging, encryption, and test harnesses. By introducing a layer for each single purpose, the design can become complicated. Therefore, it is recommended to keep the number of TAF layers low.

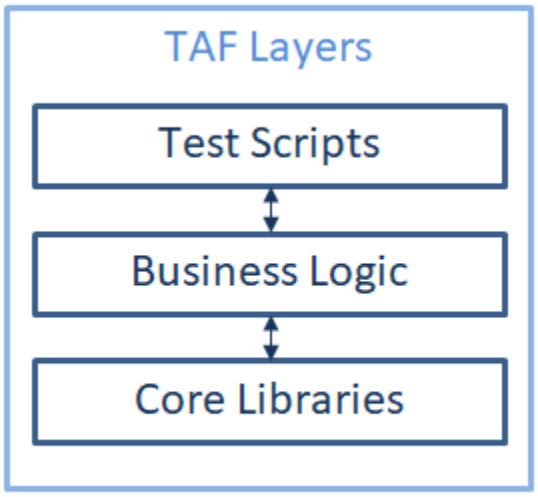

Figure 2: Test Automation Framework Layers

Test scripts

Its purpose is to provide a test case repository of the SUT and test suite annotations. It calls the services of the business logic layer which may involve test steps, user flows, or API calls. However, no direct calls should be made to the core libraries from test scripts.

Business logic

All the SUT dependent libraries are stored in this layer. These libraries will inherit the class files of the core libraries or use the facades provided by them (see section 3.1.5 regarding inheritance and facades). The business logic layer is used to set up the TAF to run against the SUT and the additional configurations.

Core libraries

All the libraries that are independent of any SUT are stored in this layer. These core libraries can be reused in any type of project that shares the same development stack.

Scaling test automation

The following example (figure 3) shows how the core libraries provide a reusable base for multiple TAFs. In Project #1 there are two TAFs built on top of the core libraries, and a separate project leverages the already existing core libraries to build their TAF for testing App #3. One TAE builds the TAFs for Project #1, while a second TAE builds the TAF for Project #2.

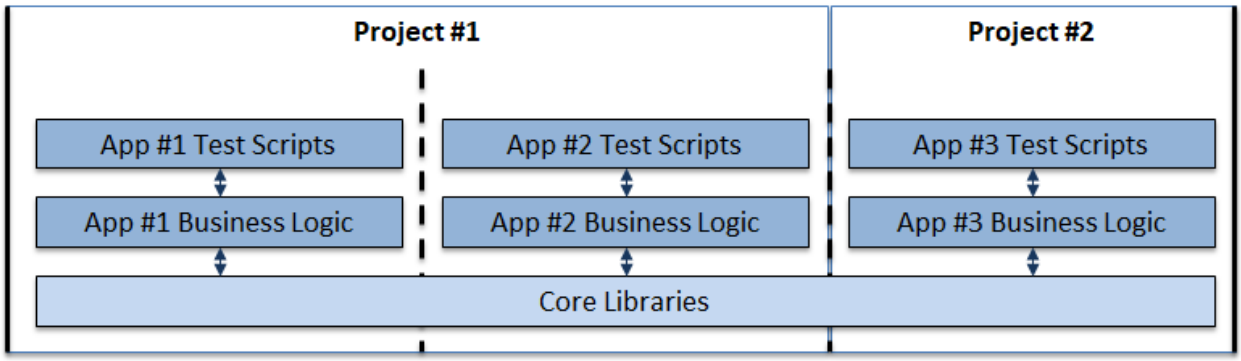

Figure 3: Application of Core Libraries to Multiple TAFs

3.1.4 Apply Different Approaches for Automating Test Cases

There are several development approaches that teams can choose from to produce automated test cases. These can include interactive scripting languages or compiled programming languages. The different approaches provide different benefits of automation and can be leveraged in different circumstances. Although test-driven development (TDD) and behavior-driven development (BDD) are development methodologies, if followed correctly, they result in automated test case development.

Capture/playback

Capture/playback is an approach that captures interactions with the SUT while a sequence of actions is performed manually. These tools produce test scripts during the capturing, and depending on the tool used, test automation code may be modifiable. The tools that do not expose code are sometimes referred to as no-code test automation, while the tools exposing code are referred to as low-code test automation.

Pros

- Initially easy to set up and use

Cons

- Hard to maintain, scale and evolve

- The SUT needs to be available while capturing a test case
- Only feasible for a small scope and an SUT that rarely changes
- The captured SUT execution depends highly on the SUT version from which the capture has been taken
- Recording each individual test case instead of reusing existing building blocks is time consuming

Linear scripting

Linear scripting is a programming activity that does not require custom test libraries made by a TAE and is used for writing and executing the test scripts. A TAE can leverage any test scripts that are recorded by a capture/playback tool, which then can be modified.

Pros

- Easy to set up and to start writing test scripts
- Compared to capture/playback, the test scripts can be modified more easily

Cons

- Hard to maintain, scale and evolve
- The SUT needs to be available while capturing a test case
- Only feasible for a small scope and an SUT that rarely changes
- Compared to capture/playback, some programming knowledge is necessary

Structured scripting

Test libraries are introduced with reusable elements, test steps and/or user journeys. Programming knowledge is necessary for the creation and maintenance of test scripts in this approach.

Pros

- Easy to maintain, scale, port, adapt and evolve
- Business logic can be separated from the test scripts

Cons

- Programming knowledge is necessary
- Initial investment into TAF development and defining the testware is time consuming

Test-driven development

Test cases are defined as part of the development process before a new feature of the SUT is implemented. The TDD approach is test, code, and refactor or otherwise known as red, green, and refactor. A developer identifies and creates one test case that will fail (red). Then he/she develops functionality that will satisfy the test case (green). The code is then refactored to optimize it and to abide by clean code principles. The process continues with the next test and next increment of functionality.

Pros

- Simplifies component level test case development
- Improves code quality and the structure of the code
- Improves testability
- Makes it easier to achieve a desired code coverage
- Reduces defect propagation to higher test levels
- Improves communication between developers, business representatives and testers
- User stories that are not verified using GUI testing and API testing can quickly achieve exit criteria by following TDD

Cons

- Initially takes more time to get accustomed to TDD
- Not following TDD properly can result in false confidence in code quality

Data-driven testing

Data-driven testing (DDT) builds upon the structured scripting approach. The test scripts are provided with test data (e.g., .csv files, .xlsx files, and database dumps). This allows for running the same test scripts multiple times with different test data.

Pros

- Allows quick and easy test case expansion through data feeds
- The cost of adding new automated tests can be significantly reduced
- Test analysts can specify automated tests by populating one or more test data files that describe the tests. This gives test analysts more freedom to specify automated tests with less dependency on the technical test analysts.

Cons

- Proper test data management may be necessary

Keyword-driven testing

Keyword-driven testing (KDT) test cases are a list or table of test steps derived from keywords and the test data which the keywords operate on. Keywords are defined from a user's perspective. This technique is often built upon DDT.

Pros

- Test analysts and business analysts can be involved in the creation of automated test cases by following the KDT approach
- KDT can also be used for manual testing independent of test automation (see the ISO/IEC/IEEE 29119-5 Standard regarding Keyword-Driven Testing)

Cons

- Implementing and maintaining keywords is a complex task that TAEs need to cover, which can become challenging when the scope grows
- It creates a huge effort for smaller systems

Behavior-Driven Development

BDD leverages a natural language format (i.e., given, when, and then) in formulating acceptance criteria that can be used as automated test cases and stored in feature files. A BDD tool then can understand the language and execute the test cases.

Pros

- Improves communication between developers, business representatives and testers
- Automated BDD scenarios act as test cases and ensure coverage of specifications
- BDD can be leveraged in producing multiple test types on different levels of the test pyramid

Cons

- Additional test cases, typically negative test conditions and edge cases still need to be defined by the team, typically by a test analyst or a TAE

- Many teams confuse BDD to be only a way of writing test cases in a natural language, and do not involve the business representatives and developers in the whole approach
- Implementing and maintaining the natural language test steps is a complex task for TAEs to cover
- Overly complex test steps will turn debugging into a difficult and costly activity

3.1.5 Apply Design Principles and Design Patterns in Test Automation

Test automation is a software development activity. Therefore, design principles and design patterns are just as important for a TAE as for a software developer.

Object-oriented programming principles

There are four major object-oriented programming principles: encapsulation, abstraction, inheritance, and polymorphism.

SOLID principles

It is an acronym of single responsibility, open-closed, Liskov substitution, interface substitution and dependency inversion. These principles improve code readability, maintainability, and scalability.

Design patterns

Of the many design patterns, three are most important for TAEs.

The facade pattern hides implementation details to only expose what the testers need to create in the test cases, and the singleton pattern is often used to make sure that there is only one driver that communicates with the SUT.

In the page object model, a class file is created and referred to as a page model. Whenever the SUT's structure changes, the TAE will have to make updates in only one place, the locator inside a page model, instead of updating the locators in each test case.

The flow model pattern is an expansion to the page object model. It introduces an additional facade over the page object models, which stores all the user actions that interact with the page objects. By introducing a double facade design, the flow model pattern provides an improved abstraction and maintainability as test steps can be reused in multiple test scripts.

4 Implementing Test Automation – 150 minutes (K4)

Keywords

risk, test fixture

Learning Objectives for Chapter 4:

4.1 Test Automation Development

TAE-4.1.1 (K3) Apply guidelines that support effective test automation pilot and deployment activities

4.2 Risks Associated with Test Automation Development

TAE-4.2.1 (K4) Analyze deployment risks and plan mitigation strategies for test automation

4.3 Test Automation Solution Maintainability

TAE-4.3.1 (K2) Explain which factors support and affect test automation solution maintainability

4.1 Test Automation Development

4.1.1 Apply Guidelines that Support Effective Test Automation Pilot and Deployment Activities

It is important to define the scope of validation for a test automation pilot. A pilot project does not take a long time to conduct, but the outcome may have a significant impact on the direction that the project takes.

Based on the information gathered about the SUT and the requirements on the project, the following should be evaluated to set up guidelines to optimize the test automation efforts:

- Programming language(s) that will be used
- Suitable commercial off-the-shelf/open-source tools
- Test levels to cover
- Test cases selected
- Test case development approach

Based on the bullet points listed above, TAEs can define an initial approach to follow. Based on the requirements, several different initial prototypes can be created to show the pros and cons of the different approaches. From there the TAEs can decide which path to move forward with.

Defining timelines is an important part of meeting schedules and ensuring the success of the pilot. A common recommendation is to periodically check on the progress of the pilot project to identify any risks and mitigate them.

During the pilot, it is also recommended to try to integrate the solution and the already implemented code into the CI/CD. This may expose issues early, either in the SUT, the TAS, or in the overall integration of different tools within the organization.

As the number of test cases grows, TAEs can think about changing the initial CI/CD setup to run the tests in different ways and at different times.

Also, during the pilot project, there is a need to evaluate other non-technical aspects, such as:

- The knowledge and experience of the team members
- The team structure
- Licensing and organization rules
- The type of planned testing and targeted test levels to cover during the automation of test cases

Once the pilot project is complete, the effort should be evaluated by TAEs and test managers to assess the success or failure and make an appropriate decision.

4.2 Risks Associated with Test Automation Development

4.2.1 Analyze Deployment Risks and Plan Mitigation Strategies for Test Automation

The interfacing of the TAF to the SUT needs to be considered as part of the architectural design. Then the packaging, test logging, and the test harness tools can be selected.

During the implementation of the pilot, expansion and maintenance of the test automation code needs to be considered. These are crucial factors of the pilot evaluation phase and can seriously affect the final decision.

Different deployment risks can be identified from the pilot:

- Firewall openings
- Resource utilization (e.g., CPU, and RAM)

Preparation must be made for deployment risks such as firewall issues, resource utilization, network connection and reliability. These are not strictly connected to test automation, but TAEs need to ensure that all conditions are met to provide reliable and beneficial quality gates in their development process.

Using real devices for mobile test automation provides an example. Mobile devices must be powered on, have enough battery power to work during the test, be connected to a network, and have access to the SUT.

Technical deployment risks can include:

- Packaging
- Logging
- Test structuring
- Updating

Packaging

Packaging needs to be considered as version control of test automation is just as important as for the SUT. Testware may need to be uploaded into a repository to share across an organization, either on the premises or in the cloud.

Logging

Test logging gives most of the information about test results. There are several test logging levels and all of them are useful in test automation for various reasons:

- Fatal: This level is used to log error events that may lead to abort the test execution
- Error: this level is used when a condition or interaction fails and therefore fails the test case as well
- Warn: This level is used when an unexpected condition/action occurs but does not break the flow of the test case
- Info: This level is used to show basic information about a test case and what happens during test execution
- Debug: This level is used to store execution specific details that generally are not required for basic logs, but useful during investigation of a test failure
- Trace: This level is similar to Debug but has even more information

Test Structuring

The most important part of the TAS is the test harness and the test fixtures included in it, items that must be available for tests to run. The test fixtures provide freedom in controlling a test environment and the test data. Preconditions and postconditions can be defined for test execution and the test cases can be grouped into test suites in several ways. These aspects are also important to evaluate during a pilot project. Moreover, the test fixtures enable the creation of automated tests that are repeatable and atomic.

Updating

One of the most common technical risks are the automatic updates on the test harnesses (e.g., agents) and version changes on devices. These risks can be mitigated by having adequate power supplies, proper network connections, and proper device configuration plans.

4.3 Test Automation Solution Maintainability

4.3.1 Explain Which Factors Support and Affect Test Automation Solution Maintainability

Maintainability is highly affected by programming standards and the TAEs' expectations of each other.
A golden rule is to try to follow the clean code principles by Robert C. Martin (Robert C Martin, "Clean Code: A Handbook of Agile Software Craftsmanship", 2008).
Briefly, clean code principles emphasize the following points:

- Use a common naming convention for the classes, methods, and variables for meaningful names
- Use a logical and common project structure
- Avoid hardcoding
- Avoid too many input parameters for methods
- Avoid long and complex methods
- Use logging
- Use design patterns where they are beneficial and required
- Focus on testability

Naming conventions are very helpful to identify the target of a given variable. Having understandable variable names such as "loginButton", "resetPasswordButton" helps TAEs to understand which component to use.

Hardcoding is the process of embedding values in the software without the ability to change them directly. It can be avoided by using data-driven testing, so the test data comes from a common source that can be maintained more easily. Hardcoding reduces development time, but it is not recommended to use as data can change frequently which can be time consuming to maintain. Using constants for those variables that are not expected to change frequently is also advised. By doing so, the sources and places that need to be maintained can be reduced.

Leveraging design patterns is also highly recommended. The design patterns - as described in 3.1.5 - enable a structured and properly maintainable test automation code to be implemented as long as the design patterns are used properly.

To ensure high quality of the test automation code, it is recommended to make use of static analyzers. Code formatters like those commonly used in IDEs will improve the readability of the test automation code.

Apart from the clean code principles, it is recommended to use an agreed branching structure and strategy in version control. Using different branches for features, releases, and defect fixes is helpful in understanding the branch contents.

5 Implementation and Deployment Strategies for Test Automation – 90 minutes (K3)

Keywords

contract testing

Learning Objectives for Chapter 5:

5.1 Integration to CI/CD Pipelines

TAE-5.1.1 (K3) Apply test automation at different test levels within pipelines

TAE-5.1.2 (K2) Explain configuration management for testware

TAE-5.1.3 (K2) Explain test automation dependencies for an API infrastructure

5.1 Integration to CI/CD Pipelines

5.1.1 Apply Test Automation at Different Test Levels within Pipelines

One of the main benefits of test automation is that the implemented tests can run unattended, making them ideal candidates to run within pipelines. This can be accomplished through CI/CD pipelines, or the pipeline used to run the tests regularly.

Test levels are usually integrated as follows:

- Configuration tests for TAF/TAS, during build can be considered as a subspecies of component tests. These tests are run during the build of a test automation project (TAF/TAS) and check that all paths to the files used in the test scripts are correct, that the files really exist and are located in the specified paths.
- Component tests are part of the build step of the pipeline, as they are executed on the individual components, (e.g., library classes, and web components). They act as quality gates for the pipeline, thus a crucial part of a continuous integration pipeline.
- Component integration tests can be part of the continuous integration pipeline if they are tests of low-level components or the SUT. In such cases, these tests and the component tests are executed together.
- System tests can often be integrated into a continuous deployment pipeline, where they act as the last quality gate of the delivered SUT.
- System integration tests between different system components are often part of a continuous delivery pipeline as quality gates. These system integration tests ensure that the separately developed system components are working together.

Many modern continuous integration systems differentiate between build and deployment phases of the continuous delivery pipelines. In these cases, component tests and component integration tests are part of the first build phase. When this first phase is successful (i.e., build and test together), the components/SUT are deployed.

In case of system integration testing, system testing and acceptance testing, there are two main approaches to integrate them into such pipelines:

1. Test cases are executed as part of the deployment phase after the component deployment. This can be beneficial, as based on the test results, the deployment can fail and also be rolled back. However, in this case, if tests need to be rerun, a redeployment needs to be done.
2. Test cases are executed as a separate pipeline, triggered by the successful deployment. This can be beneficial if it is expected that different test suites and various test automation code will run on each deployment. In this case, tests do not act as a quality gate. Thus, it requires other, usually manual, actions to roll back an unsuccessful deployment.
   In this case, a few simple automated test scripts, as deployment checks, can be used to ensure the SUT is deployed, but these automated test scripts do not verify functional suitability in a broad manner.

Pipelines can also be used for other test automation purposes, such as:

- Running different test suites periodically: A regression test suite can be run every night (i.e., the nightly regression), especially for longer running test suites, so the team will have a clear picture of the quality of the SUT in the morning.
- Running non-functional tests: Either part of a continuous deployment pipeline, or separately, to periodically monitor certain non-functional quality characteristics of the system such as performance efficiency

5.1.2 Explain Configuration Management for Testware

Configuration management is an integral part of test automation, as automation will often be executed on multiple test environments and versions of the SUT.

Configuration management in test automation includes:

- Test environment configuration
- Test data
- Test suites/test cases

Test environment configuration

Each test environment used in the development pipeline can have different configurations, such as various URLs or credentials. The test environment configuration is usually stored with the testware. However, in the case of test automation used on multiple projects or multiple TAFs for the same project, the test environment configuration can be part of the common core library or in a shared repository.

Test data

Test data can also be specific for the test environment or for the release and the feature set of the SUT. As with the test environment configuration, test data is usually stored with smaller TAFs, but test data management systems can also be used.

Test suites/test cases

A common practice is to set up different test suites of the test cases, based on their purpose, such as smoke testing or regression testing. These test suites are often executed in separate test levels, leveraging different pipelines and test environments.

Each release of the SUT determines a feature set which includes test cases and test suites that assess the quality of the given release. There are different options in the testware to handle this:

- A feature toggle configuration can be defined per each release or test environment. There are test cases and test suites to test each feature. The feature toggle can be used in the testware to identify which test suites to execute on a given release/test environment.
- The testware can also be released with the SUT using the same release version. In this way, there is an exact match between the SUT version and the testware that can test it. Such a release is usually implemented using a configuration management system using tags or branches.

5.1.3 Explain Test Automation Dependencies for an API Infrastructure

When performing API test automation, it is crucial to have the following information about dependencies to build a proper strategy:

- API connections: Understand the business logic that can be tested automatically and the relationship between APIs
- API documentation: Serves as a baseline for test automation with all relevant information (e.g., parameters, headers, and distinct types of request-response of objects)

Integrated automated API testing can be done by either the developers or the TAEs. However, with shift left it is recommended to support and divide the testing among different levels. In the ISTQB CTFL Syllabus, component integration testing and system integration testing are mentioned which can be extended with a best practice called contract testing.

Contract testing

Contract testing is a type of integration testing verifying that services can communicate with each other, and that the data shared between the services is consistent with a specified set of rules. Using contract testing provides compatibility across two separate systems (e.g., two microservices) to communicate with one another. It goes beyond schema validation, requiring both parties to come to a consensus on the allowed set of interactions while providing for evolution over time. It captures the interactions that are exchanged between each service, storing them in a contract, which can then be used to verify that both parties adhere to it. One of the main advantages of this test type is that defects occurring from underlying services can be found earlier in the SDLC and the source of these defects can be more easily identified.

In the consumer-driven approach to contract testing, the consumer sets its expectation determining how the provider shall respond to requests coming from this consumer. In the provider-driven approach to contract testing, the provider creates the contract, which shows how its services are operating.

6 Test Automation Reporting and Metrics – 150 minutes (K4)

Keywords

measurement, metric, test log, test progress report, test step

Learning Objectives for Chapter 6:

6.1 Collection, Analysis and Reporting of Test Automation Data

TAE-6.1.1 (K3) Apply data collection methods from the test automation solution and the system under test

TAE-6.1.2 (K4) Analyze data from the test automation solution and the system under test to better understand results

TAE-6.1.3 (K2) Explain how a test progress report is constructed and published

6.1 Collection, Analysis and Reporting of Test Automation Data

6.1.1 Apply Data Collection Methods from the Test Automation Solution and the System Under Test

Data can be collected from the following sources:

- SUT logs
  - Web/mobile UI
  - APIs
  - Applications
  - Web servers
  - Database servers
- TAF logs to provide an audit trail
- Build logs
- Deployment logs
- Production logs to monitoring data in production (see the ISTQB CT-PT Syllabus, section 2.3)
  - Performance monitoring in production to perform trend analysis
  - Performance efficiency test logs in a performance test environment (e.g., load, stress, and spike testing)
- Screenshots and screen recordings (native to the automation tool or 3rd party)

Since a TAS has automated testware at its core, the automated testware can be enhanced to record information about its use. Testware enhancements made to the underlying testware can be used by all the higher-level automated test scripts. For example, enhancing the underlying testware to record the start and end time of test execution may well apply to all tests.

Features of test automation that support measurement and test report generation

The scripting languages of many test tools support measurement and reporting through facilities that can be used to record and log information before, during, and after test execution of individual tests, and entire test suites.

Test reporting on each of a series of test runs needs to have an analysis feature to consider the test results of the previous test runs so it can highlight trends, such as changes in the test success rate.

Test automation typically requires automating both the test execution and the test verification, the latter being achieved by comparing specific elements of the actual results with the expected results. This comparison is best done by a test tool using assertions. The level of information that is reported as a result of this comparison must be considered. It is important that the test status be determined correctly (i.e., passed or failed). In the case of failed status, more information about the cause of the failure will be required (e.g., screen shots).

Differences between actual results and expected results of a test are not always clear, and tool support can help greatly in defining comparisons that ignore differences that are expected, such as dates and times, while highlighting any unexpected differences.

Test logging

Test logs are a source that is frequently used to analyze potential defects within the TAS and the SUT. In the following section are examples of test logging, categorized by TAS and SUT.

TAS logging

The context determines whether the TAF or the test execution is responsible for logging information that should include the following:

- Which test case is currently under test execution, including start and end time
- The status of the test execution because, while failures can easily be identified in test logs, the TAF should also have this information and should report via a dashboard. The test execution status can be either passed, failed, or a TAS failure. Status can sometimes be inconclusive and it is important for an organization to define clear and consistent definitions for them. A TAS failure is applied to situations where the defect is not in the SUT.
- Low-level details of the test log (e.g., logging significant test steps) including timing information
- Dynamic information about the SUT (e.g., memory leaks) that the test case was able to identify with the help of third-party tools. Actual results and failures should be logged with the test suite that was executed when the failure was detected.
- In the case of reliability testing or stress testing where numerous test cycles are performed, a counter should be logged to easily determine how many times test cases have been executed.
- When test cases have random elements (e.g., random parameters or random test steps in state transition testing), the random number/choices should be logged.
- All actions that a test case performs should be logged in such a way that the test logs, or parts of it, can be played back to re-run the test with the same test steps and the same timing. This is useful to reproduce an identified failure and to capture additional information. The test case action information can also be logged by the SUT for use when reproducing customer identified failures. If a customer runs a test suite, the test log information is captured and can then be replayed by the development team when troubleshooting a defect.
- Screenshots can be saved during test execution for further use during root cause analysis.
- Whenever a test suite initiates a failure, the TAS should make sure that all information needed to analyze the defect is available/stored, as well as any information regarding the continuation of testing, if applicable. The TAS should save any associated crash dumps and stack traces. Also, any test logs which can be overwritten (e.g., cyclic buffers are often used for test logs on the SUT) should be stored where they will be available for later analysis.
- Use of color can help to distinguish different types of test log information (e.g., defects in red, and progress information in green).

SUT logging

Correlation of the test automation results with SUT logs to helps identify the root cause of defects in the SUT and the TAS.

- When a defect is identified in the SUT, all necessary information needed to analyze the defect should be logged, including date and time stamps, source location of the defect, and error messages.
- At the startup of a system, configuration information should be logged to a file, consisting of, for example, the different software/firmware versions, configuration of the SUT, and configuration of the operating system.
- Using test automation, SUT logs can be easily searchable. A failure identified in the test log by the TAS should be easily identified in the test log of the SUT, and vice versa, with or without additional tools. Synchronizing various test logs with a time stamp facilitates correlation of what occurred when a failure is reported.

Integration with other third-party tools (e.g., spreadsheets, XML, documents, databases, and report tools)

When information from the execution of automated test cases is used in other tools for tracking and reporting (e.g., updating traceability information), it is possible to provide the information in a format that is suitable for third-party tools. This is often achieved through existing test tool functionality (e.g., export formats for test reporting) or by creating customized reporting that is output in a format consistent with other software.

Visualization of test results

Test results can be made visible using charts. Consider using colored icons such as traffic lights to indicate the overall status of the test execution/test automation so that decisions can be made based on reported information. Management is particularly interested in visual summaries to see the test results, which aides in decision making. If more information is needed, they can still drill down into the details.

6.1.2 Analyze Data from the Test Automation Solution and the System Under Test to Better Understand Test Results

After test execution it is important to analyze the test results, to identify possible failure(s) both in the SUT and the TAS. For such an analysis the data collected from the TAS is primary and the data collected from the SUT is secondary.

- Analyze the test environment data to support proper sizing of test automation (e.g., in the cloud)
  - Clusters and resources (e.g., CPU, and RAM)
  - Single versus multi browser (i.e., cross-browser) test execution
- Compare test results of previous test executions
- Determine how to use web logs to monitor software usage

Test execution failures need to be analyzed, as there are potential issues:

1. Check if the same failure happened in the previous test executions. This might be a known defect either in the SUT or the TAS. The TAS can be built to log historical test results of the test cases, thus further helping the analysis.
2. If the defect is not known, identify the test case and what it is testing. The test can be self-explanatory, or the test case can be identified in the test management system, based on its ID logged with the test execution.
3. Find in which test step of the test case the failure happened. The TAS logs this.
4. Analyze the test log information about the state of the SUT and whether it matches the expected results using screenshots, API and network logs, or any log that shows the state of the SUT.
5. If the state of the SUT is not what was expected, log a defect in the defect management system. Make sure to include all the necessary defect information and the logs that justify that it is a defect.

When there is a failure, it is possible for the actual result and the expected result of the SUT to match. In this case, most likely the TAS contains a defect which needs to be fixed, or an invisible mismatch is present.

Another situation that can occur is if the test environment is not available during the test run, or only partially available. In this case, all test cases can fail, either with the same defect or if parts of the system are down, with seemingly real failures. To identify the root cause of such defects, the SUT logs can be analyzed, which will show if there were any test environment outages at the time of the test run.

If the SUT implements audit logs for user interactions (i.e., UI sessions or API calls) it helps to analyze test results. There is usually a unique ID added to the interaction with the same ID for each subsequent call and integration in the system. In this way, knowing the unique ID of a request/interaction, the behavior of the system can be observed and traced back.

This unique ID is usually called a correlation ID or trace ID. This can be logged by the TAS to help analyze test results.

6.1.3 Explain How a Test Progress Report is Constructed and Published

The test logs give detailed information about the test steps, actions to take, and expected responses of a test case and/or test suite. However, the test logs alone cannot provide a good overview of the overall test results. For this, it is necessary to have test reporting functionality. After the execution of a test suite, a concise test progress report must be created and published. A report generator can be used for this.

Content of a test progress report

The test progress report must contain the test results, SUT information, and documentation of the test environment in which the tests were run in a format appropriate for each of the stakeholders.

It is necessary to know which tests have failed and the reasons for failure. To make troubleshooting easier, it is important to know the test execution history and who reported it (i.e., generally the person who created or last updated it). The person responsible needs to investigate the cause of the failure, report the defect related to it, follow-up on the fix to the defect and test that the fix has been correctly implemented.

Test reporting is also used to diagnose any failures of the TAF components.

Publishing the test reports

The test report should be published to all relevant stakeholders. It can be uploaded on a website, in the cloud or on the premises, sent to a mailing list or uploaded to another tool such as a test management tool. This helps ensure that reports will be reviewed and analyzed if individuals are subscribed to receive them by email or through chat messages posted by a chatbot.

An option is to identify problematic parts of the SUT, and keep a history of the test reports, so that statistics about test cases or test suites with frequent regressions can be gathered for trend analysis.

Stakeholders to report to include:

- Management stakeholders
  - Typical roles: solution or enterprise architect, project/delivery manager, program manager, test manager, or test director
- Operational stakeholders
  - Typical roles: product owner/manager, business representative, or business analyst
- Technical stakeholders
  - Typical roles: team leader, scrum master, web administrator, database developer/administrator, test leader, TAE, tester, or developer

Test reports may vary in content or detail depending on the recipients. While technical stakeholders may be more interested in lower-level details, management will focus on trends, such as how many test cases were added since the last test run, changes in the pass-fail ratio and the reliability of the TAS and SUT. Operational stakeholders usually put more emphasis on product use related metrics.

Creation of dashboards

Modern reporting tools provide several reporting options through dashboards, colorful charts, detailed log collections and automated test log analysis. There are many available tools in the market to choose from.

These tools support data aggregation from sources such as pipeline execution test logs, project management tools, and code repositories. The visualization of data provided by these tools helps stakeholders see trends and make decisions accordingly. These trends can include defect clusters, increase/decrease in defect propagation to certain test environments, SUT performance degradation, and reliability of builds.

Artificial Intelligence/machine learning analysis of test logs

In recent years, some test automation tools include or are based on machine learning (ML) algorithms. Automated analysis of large amounts of data in test logs helps the TAE reduce time spent finding broken locators, analyzing the reason for test failures (i.e., is it a defect in the SUT or in the TAS?) and grouping common defects for test reporting (see the ISTQB CT-AI Syllabus).

7 Verifying the Test Automation Solution – 135 minutes (K3)

Keywords

static analysis

Learning Objectives for Chapter 7:

7.1 Verification of the Test Automation Infrastructure

TAE-7.1.1 (K3) Plan to verify the test automation environment including test tool setup

TAE-7.1.2 (K2) Explain the correct behavior for a given automated test script and/or test suite

TAE-7.1.3 (K2) Identify where test automation produces unexpected results

TAE-7.1.4 (K2) Explain how static analysis can aid test automation code quality

7.1 Verification of the Test Automation Infrastructure

7.1.1 Plan to Verify the Test Automation Environment Including Test Tool Setup

Whether the test automation environment and all of the other components of the TAS work as expected still requires verification. These checks are done, for example, before starting test automation. Steps can be taken to verify the components of the test automation environment. Each of these is explained in more detail below.

Test tool installation, setup, configuration, and customization

The TAS is comprised of many components. Each of these needs to be counted on to ensure reliable and repeatable performance. At the core of a TAS are the executable components, corresponding function libraries, and supporting data and configuration files. The process of configuring a TAS may range from the use of automated installation scripts to manually placing files in corresponding folders. Test tools, similar to operating systems and other software, regularly have service packs or may have optional or required add-ins to ensure compatibility with any given SUT environment.

Automated installation or copying from a repository has advantages. It can guarantee that tests on different SUTs have been performed with the same version of the TAS, and the same configuration of the TAS, where this is appropriate. Upgrades to the TAS can be made through the repository. Repository usage and the process to upgrade to a new version of the TAS should be the same as that for standard development tools.

Repeatability in setup/teardown of the test environment

A TAS will be implemented on a variety of systems, servers, and to support CI/CD pipelines. To ensure that the TAS works properly in each test environment, it is necessary to have a systematic approach to loading and unloading the TAS from any given test environment. This is successfully achieved when the building and rebuilding of the TAS provides no discernible difference in how it operates within and across multiple test environments. Configuration management of the TAS components ensures that a given configuration can be dependably created. Once this is accomplished, documenting the various components that comprise the TAS will provide the necessary knowledge for what aspects of the TAS may be affected or require change when the SUT environment changes.

Connectivity with internal and external systems/interfaces

Once a TAS is installed in a given SUT environment, and prior to using the SUT, a set of checks or preconditions should be administered to ensure that connectivity to internal systems, external systems, and interfaces are available. For example, it is a good practice to log into servers, launch test automation tools, verify that the test automation tools can access the SUT, manually inspect configuration settings, and ensure that permissions are set properly for test logging and test reporting between systems. Establishing preconditions for test automation is essential in ensuring that the TAS has been installed and configured correctly.

TAF component testing

Just like any software development project, the TAF components need to be individually tested and verified. This may include functional and non-functional testing (e.g., performance efficiency, and resource utilization). For example, components that provide object verification on GUI systems need to be tested for a wide range of object classes to establish that the object verification functions correctly. Likewise, test logs and test reports should produce accurate information regarding the status of test automation and SUT behavior. Examples of non-functional testing may include understanding TAF performance degradation, utilization of system resources that may indicate defects such as memory leaks, and a lack of interoperability of components within and/or outside of the TAF.

7.1.2 Explain the Correct Behavior for a Given Automated Test Script and/or Test Suite

Automated test suites need to be tested for completeness, consistency, and correct behavior. Different kinds of verification checks can be applied to make sure the automated test suite is available at any given time, or to determine that it is fit for use.

Steps can be taken to verify the automated test suite. These include:

- Check the composition of the test suite
- Verify new tests that focus on new features of the TAF
- Consider the repeatability of tests
- Consider the intrusiveness of automated test tools

Each of these is explained in more detail below.

Check the composition of the test suite

Check for completeness (e.g., test cases all have expected results, and test data is present), and for the correct version of the TAF and the SUT.

Verifying new tests that focus on new features of the TAF

The first time a new feature of the TAF is used in test cases, it should be verified and monitored closely to ensure the feature is working correctly.

Consider the repeatability of tests

When repeating tests, the test results should always be the same. Having test cases in the test suite which do not give a reliable test result (e.g., due to race conditions) should be moved from the active automated test suite and analyzed separately to find the root cause. Otherwise, time will be spent repeatedly on these test runs to analyze the failure.

Consider the intrusiveness of automated test tools

The TAS will often be tightly coupled with the SUT. This is by design so that there is better compatibility as it pertains to the level of interactions. However, this close integration can also lead to adverse outcomes. For instance, when the TAS is located within the SUT environment, the SUT's functionality may differ from when tests are conducted manually, potentially impacting performance as well.

A high level of intrusion can show failures during testing that are not evident in production. If this causes failures with the automated tests, the confidence in the TAS can drop dramatically. Developers may require that failures identified by test automation be reproduced manually, if possible, to assist with the analysis.

7.1.3 Identify Where Test Automation Produces Unexpected Results

When a test script fails or passes unexpectedly, root cause analysis must be performed. This will include inspecting test logs, performance data, setup, and teardown of the test script.

It is also helpful to execute a few isolated tests. Intermittent failures are more difficult to analyze. The defect can be in the test case, the SUT, the TAF, the hardware or the network. Monitoring system resources may yield clues for the root cause. Test log file analysis of the test case, the SUT and the TAF can help identify the root cause of the defect. Debugging may also be necessary. To aid in identifying the root cause may require support from a test analyst, business analyst, developer, or system engineer.

Verify if all the assertions are in place. Missing assertions may result in inconclusive test results.

7.1.4 Explain How Static Analysis Can Aid Test Automation Code Quality

Static code analysis can help find vulnerabilities and defects in program code. This can include the SUT or the TAF.

Automated scans can inspect code to mitigate risks. This provides a review of the SUT for defects and ensures code quality standards are being met and applied. This can also be considered a proactive defect detection technique, and this plays a significant role in DevSecOps implementations (i.e., DevOps with emphasis on Security). These scans occur early in the SDLC via pipelines to provide development teams with immediate feedback.

The output regarding defects is usually categorized as critical, high, medium, or low severity so that development teams have the ability to prioritize which defects they choose to fix. Certain static analysis tools also have the ability to provide suggested code fixes to address the defects that they find. It will present development teams with a copy of the offending lines of code and offer a possible remedy for developers to implement. Additionally, these tools aid TAEs by measuring quality, suggesting areas where to comment code, improving code design for optimized resource handling (e.g., utilizing try/catch blocks, and better looping structures), and removing poor library calls.

As test automation tools use programming languages there is a risk that inadequate test automation code can be introduced to the SDLC. As a simple example, a common practice when automating tests is to have a username and password. It is conceivable that a TAE can incorrectly include the password in plaintext within one or many test scripts.

Static analysis tools can benefit test automation code. They can be used to analyze the test automation code for security violations such as a plaintext password within the code. Static analysis tools support many programming languages, including those used by test automation software. Therefore, it is imperative that the TAE extends the best practices of scanning code to also include test automation code. Even though the test automation code is not necessarily deployed with the overall software, there are clearly potential vulnerabilities if the password was discovered in an accompanying automation test script (see the ISTQB CT-SEC Syllabus).

8 Continuous Improvement – 210 minutes (K4)

Keywords

schema validation, test histogram

Learning Objectives for Chapter 8:

8.1 Continuous Improvement Opportunities for Test Automation

TAE-8.1.1 (K3) Discover opportunities for improving test cases through data collection and analysis

TAE-8.1.2 (K4) Analyze the technical aspects of a deployed test automation solution and provide recommendations for improvement

TAE-8.1.3 (K3) Restructure the automated testware to align with SUT updates

TAE-8.1.4 (K2) Summarize opportunities for use of test automation tools

8.1 Continuous Improvement Opportunities for Test Automation

8.1.1 Discover Opportunities for Improving Test Cases Through Data Collection and Analysis

Data collection and analysis can be improved when considering various types of data with the approaches described below.

Test histogram

A visual report of test data represented as a test histogram provides potential improvement areas regarding test case data trends. TAEs can decide on possible improvement areas since many CI/CD and test reporting tools have the capability to show different test results and their respective test data (e.g., exception logs, error messages, and screenshots). The test histogram also enables the TAEs to identify and select those test cases that are fragile and refactor them with additional improvements or by rethinking the actual implementation.

Artificial Intelligence

Another recent opportunity is utilizing Artificial Intelligence (AI) to support testing and test automation. For example, in UI test cases, data also includes UI locator values that can be treated as inputs. Recent cutting-edge tools show the capability of detecting if a given locator is changed from the one used. Based on ML and image recognition, they can identify the new selectors and use a self-healing algorithm to fix the test case and include the changed locators in the test report. This can speed up the follow-up steps such as version control changes and code maintenance.

Schema validation

Schema validation can be applied in API data analysis (e.g., properties derived from target endpoints) and database analysis (e.g., validation rules for software fields, such as allowed data types and value ranges).

With schema validation, the TAS is capable of checking if a response matches the actual business specification. This kind of check can be used to determine if mandatory response elements are present in the service response and if their object type matches the defined one in the schema. In case of the schema breaking, the solution returns the actual validation that helps the TAEs to identify the root cause of the problem.

Example: an API has six mandatory response elements that must be strings in the response. With schema validation tools it is not required to write individual assertions to check if these types are strings, and their values are not null. The schema validation will handle these checks, making the implemented test automation code much shorter and increasing the efficiency of detecting defects in the backend service.

8.1.2 Analyze the Technical Aspects of a Deployed Test Automation Solution and Provide Recommendations for Improvement

In addition to the ongoing maintenance tasks that are necessary to keep the TAS synchronized with the SUT, there are many opportunities to improve the TAS. These improvements may be made to achieve a range of benefits including greater efficiency (e.g., further reducing manual intervention), better ease of use, additional capabilities, and improved support for testing. The decision as to how the TAS is improved is influenced by what features add the most value to a project.

Specific areas of a TAS that may be considered for improvement include scripting, test execution, verification, the TAA, the TAF, setup and teardown, documentation, TAS features, and TAS updates and upgrades. These are described in more detail below.

Scripting

Scripting techniques vary from the linear scripting to the data-driven testing approach and on to the more sophisticated keyword-driven testing approach, as described in section 3.1.4. It may be appropriate to upgrade the current TAS scripting technique for all new automated tests. The technique may be retrofitted to all existing automated tests or at least those that involve the greatest amount of maintenance effort.

Another area of TAS improvement for test scripts may focus on their implementation. For example:

- Assess test script/test case/test step overlap to consolidate automated tests. Test cases containing similar action sequences should not implement these test steps multiple times. These test steps should be made into a function and added to a library, so that they can be reused. These library functions can then be used by different test cases. This increases the maintainability of the testware. When test steps are not identical but similar, parameterization may be necessary. Note: this is a typical approach in keyword-driven testing.
- Establish a failure recovery process for the TAS and the SUT. When a failure occurs during the execution of a test suite, the TAS should be able to recover from this condition to continue with the next possible test. When a failure occurs in the SUT, the TAS needs to perform necessary recovery actions on the SUT (e.g., a reboot of the SUT) where feasible and practical.
- Evaluate wait mechanisms to ensure the best type is being used. There are three common wait mechanisms:
    - Hard coded waits (i.e., wait a certain number of milliseconds) which can be a root cause for many test automation defects given the unpredictability of software response times.
    - Dynamic waiting by polling (e.g., checking that a certain state change or action has taken place) is much more flexible and efficient:
        - The TAS waits only the needed amount of time, and no test time is wasted
        - When the process takes longer than expected, the polling will wait until the condition is true. Remember to include a timeout mechanism, otherwise the test may wait forever if there is a defect.
    - An even better way is to subscribe to the event mechanism of the SUT. This is much more reliable than the other two options, but the test scripting language needs to support event subscription and the SUT needs to offer these events to the TAS. A timeout mechanism is also required, or the test may wait forever if there is a defect.

Test execution

When an automated regression test suite is not finished because test execution takes too long, it may be necessary to test concurrently on different test environments if this is possible. When expensive systems are used for testing, it can be a constraint that all testing must be done on a single target system. It may be necessary to split the regression test suite into multiple parts, each executing in a defined period of time (e.g., in a single night). Further analysis of the test automation coverage may reveal duplication. Removing duplication can reduce test execution time and can yield further efficiencies. In the case of CI/CD, a good practice is to run batch jobs in parallel to optimize the test execution time. Also, it is good practice to schedule automated batch jobs to run the different pipelines at a given time, e.g., every morning to reduce the manual interactions and speed up the development process.

Verification

Before creating new verification functions, adopt a set of standard verification methods for use by all automated tests. This will avoid the re-implementation of verification actions across multiple tests. When verification methods are not identical but similar, the use of parameterization will aid in allowing a function to be used across multiple types of objects.

TAA

It may be necessary to change the TAA to support improvements of the testability of the SUT. These changes may be made in the architecture of the SUT and/or in the TAA of the TAS. This can provide a major improvement in the test automation, but may require significant changes and investment in the SUT/TAS. For example, if the SUT is going to be changed to provide APIs for testing then the TAS should also be refactored accordingly. Adding these kinds of features later in the SDLC can be quite expensive; it is much better to think about this at the start of test automation and in the early phases of the SDLC of the SUT.

TAF

Often, there are new versions of the core libraries used in a TAF. Sometimes these are major updates, and the latest version cannot be immediately referenced in the TAF's dependency list as it would break the tests for many of the teams using them. Therefore, it is better to first perform a pilot and an impact analysis. After that, an adoption plan can be created. Either all the teams adopt the new version of the core libraries at the same time by updating the dependency in the core libraries layer's build file, or each team individually decides when to update it in their business logic layer. Eventually, once all the teams are ready to accept the new version of the core libraries, then the dependencies can be updated in the core libraries layer (see section 3.1.3).

Setup and teardown

Actions and configurations that are repeated before or after each test script or test suite should be moved into setup or teardown methods. This way any changes impacting the code can be updated in one place, which results in decreased maintenance efforts. For example, web service calls can be used to fulfill preconditions or postconditions for UI tests (e.g., user registration, user cleanup, and profile setup).

Documentation

This covers all forms of documentation from test automation documentation (e.g., what the test automation code does, and how it should be used), user documentation for the TAS, and the test reports and test logs produced by the TAS.

TAS features

Add TAS features and functions such as detailed test reporting, test logs, and integration to other systems. Only add new features that will be used. Adding unused features only increases complexity and decreases reliability and maintainability.

TAF updates and upgrades

By updating or upgrading to new versions of the TAF, new functions may become available that can be used by the test cases or failures may be corrected. The risk is that updating the TAF by either upgrading the existing test tools or introducing new ones might have an adverse impact on the existing test cases. Test the latest version of the test tool by running sample tests before rolling out the new version of the test tool. The sample tests should be representative of the automated tests of different SUTs, different test types and, where appropriate, different test environments.

8.1.3 Restructure the Automated Testware to Align with System Under Test Updates

Following a given set of changes to an existing SUT will require updates to the TAS including the TAF and component libraries. Any change, no matter how trivial, may have a wide-ranging adverse impact on the reliability and performance of the TAS.

Identify changes in the test environment components

Evaluate what changes and improvements need to be made. Do these require changes to the testware, the customized function libraries, or the operating system? Each of these has an impact on how the TAS performs. The overall goal is to ensure automated tests continue to run in an efficient manner. Changes should be made incrementally, with a minimum viable product mindset, so that the impact on the TAS can be measured through a limited run of test scripts. Once it is found that no side effect exists, changes can be fully implemented. A full regression run is the last step toward verifying that the change did not adversely affect the automated test scripts. During execution of these regression test scripts, failures may be found. Identifying the root cause of these failures (e.g., through test reporting, test logs, and test data analysis) will provide a means to ensure they do not result from the test automation improvement activity.

Increase efficiency and effectiveness of core TAS function libraries

As a TAS matures, new ways are discovered to perform tasks more efficiently. These new techniques (e.g., optimizing code in functions, and using newer operating system libraries) need to be incorporated into the core function libraries that are used by the current project and future projects.

Target multiple functions that act on the same control type for consolidation

A large part of what occurs during an automated test run is the interrogation of controls in the GUI. This interrogation serves to provide information about a control (e.g., visible/not visible, enabled/not enabled, size and dimensions, and data). With this information, an automated test can select an item from a dropdown list, enter data into a field, and read a value from a field. There are several functions that can act upon controls to elicit this information. Some functions are extremely specialized, while others are more general in nature. For example, there may be a specific function that works only on dropdown lists. Alternatively, there may be a function that works with several functions by specifying a function as one of its parameters. Therefore, a TAE may use several functions that can be consolidated into fewer functions, achieving the same results, and minimizing maintenance.

Refactor the TAA to accommodate changes in the SUT

Through the lifecycle of a TAS, changes will need to be made to accommodate changes in the SUT. As the SUT evolves and matures, the underlying TAA will have to evolve as well to ensure that the capability is there to support the SUT. Care must be taken when extending features so that they are not implemented in a bolt-on manner, but instead are analyzed and changed in the TAA. This will ensure that, as new SUT functionality requires additional test scripts, compatible components will be in place to accommodate these new automated tests.

Naming conventions and standardization

As changes are introduced, naming conventions for new test automation code and function libraries need to be consistent with previously defined standards (see section 4.3.1).

Evaluation of existing test scripts for SUT revision/elimination

The process of change and improvement also includes an assessment of existing test scripts, their use and continued value. For example, if certain tests are complex and time consuming to run, decomposing them into smaller tests can be more viable and efficient. Targeting tests that run infrequently or not at all for elimination will pare down the complexity of the TAS and bring greater clarity to what needs to be maintained.

8.1.4 Summarize Opportunities for Use of Test Automation Tools

Apart from the actual testing, test automation can help in nonspecific test activities such as:

Environment setup and control

Certain test scripts (e.g., test data creation) can be leveraged in a setup method to create different test data in a new test environment. In a situation where users with multiple profiles need to be created based on different data inputs, a team can use an automated test script to call a web service endpoint that registers these users. Test scripts can have control over the setup of test infrastructure and can be leveraged in a cleanup following the process. This helps save time and ensure that the proper users are present in each new test environment. As an example, different test logs and other testware can be removed from a test environment automatically, making the housekeeping and the use of the test environment more efficient.

Data aging

Test automation can be used to manipulate the test data in the test environment. For example, in databases, the date fields can be checked and controlled to keep them up to date from a year perspective.

Screenshot and video generation

Most modern UI test automation tools have a built-in capability to create screenshots or videos on certain conditions and to store them. With these test tools, the teams can support the business to create actual usage screenshots and videos for software release documentation or marketing purposes.

9 References

Standards

Standards for test automation include but are not limited to:

The Automatic Test Markup Language (ATML) by IEEE (Institute of Electrical and Electronics Engineers) consisting of ● IEEE Std 1671.1: Test Description ● IEEE Std 1671.2: Instrument Description ● IEEE Std 1671.3: UUT Description ● IEEE Std 1671.4: Test Configuration Description ● IEEE Std 1671.5: Test Adaptor Description ● IEEE Std 1671.6: Test Station Description ● IEEE Std 1641: Signal and Test Definition IEEE Std 1636.1: Test Results
ISO/IEC/IEEE 29119-5 (2016) Software and systems engineering – Software testing – Part 5: Keyword-Driven Testing
ISO/IEC 30130:2016 (E) Software engineering — Capabilities of software testing tools The Testing and Test Control Notation (TTCN-3) by ETSI (European Telecommunication Standards Institute) and ITU (International Telecommunication Union) consisting of ● ES 201 873-1: TTCN-3 Core Language ● ES 201 873-2: TTCN-3 Tabular Presentation Format (TFT) ● ES 201 873-3: TTCN-3 Graphical Presentation Format (GFT) ● ES 201 873-4: TTCN-3 Operational Semantics ● ES 201 873-5: TTCN-3 Runtime Interface (TRI) ● ES 201 873-6: TTCN-3 Control Interface (TCI) ● ES 201 873-7: Using ASN.1 with TTCN-3 ● ES 201 873-8: Using IDL with TTCN-3 ● ES 201 873-9: Using XML with TTCN-3 ● ES 201 873-10: TTCN-3 Documentation ● ES 202 781: Extensions: Configuration and Deployment Support ● ES 202 782: Extensions: TTCN-3 Performance and Real-Time Testing ● ES 202 784: Extensions: Advanced Parameterization ● ES 202 785: Extensions: Behaviour Types ● ES 202 786: Extensions: Support of interfaces with continuous signals ● ES 202 789: Extensions: Extended TRI
The UML Testing Profile (UTP) by OMG (Object Management Group) specifying test specification concepts for ● Test Architecture ● Test Data ● Test Behavior ● Test Logging ● Test Management

ISTQB® Documents

Identifier	Reference
ISTQB-AL-TTA	ISTQB Certified Tester, Advanced Level Syllabus, Technical Test Analyst, Version 4.0, June 2021, available from [ISTQB-Web]
ISTQB-FL	ISTQB Certified Tester, Foundation Level Syllabus, Version 4.0, April 2023, available from [ISTQB-Web]
ISTQB-MBT	ISTQB Certified Tester, Model-Based Testing Syllabus, Version 1.0, October 2015, available from [ISTQB-Web]
ISTQB-PT	ISTQB Certified Tester, Performance Testing Syllabus, December 2018, available from [ISTQB-Web]
ISTQB-SEC	ISTQB Certified Tester, Security Tester Syllabus, Version 1.0, March 2016, available from [ISTQB-Web]
ISTQB-TAS	ISTQB Certified Tester, Test Automation Strategy Syllabus, February 2024, available from [ISTQB-Web]
ISTQB-Glossary	ISTQB Glossary of Terms, available online from [ISTQB-Web]

Books

Paul Baker, Zhen Ru Dai, Jens Grabowski, and Ina Schieferdecker, “Model-Driven Testing: Using the UML Testing Profile”, Springer 2008 edition, ISBN-10: 3540725628, ISBN-13: 978-3540725626
Efriede Dustin, Thom Garrett, Bernie Gauf, “Implementing Automated Software Testing: how to save time and lower costs while raising quality”, Addison-Wesley, 2009, ISBN 0-321-58051-6
Efriede Dustin, Jeff Rashka, John Paul, “Automated Software Testing: introduction, management, and performance”, Addison-Wesley, 1999, ISBN-10: 0201432870, ISBN-13: 9780201432879
Mark Fewster, Dorothy Graham, “Experiences of Test Automation: Case Studies of Software Test Automation”, Addison-Wesley, 2012
Mark Fewster, Dorothy Graham, “Software Test Automation: Effective use of test execution tools”, ACM Press Books, 1999, ISBN-10: 0201331403, ISBN-13: 9780201331400
Robert C Martin, “Clean Code: A Handbook of Agile Software Craftsmanship”, 2008, ISBN-10: 9780132350884
James D. McCaffrey, “.NET Test Automation Recipes: A Problem-Solution Approach”, APRESS, 2006 ISBN-13:978-1-59059-663-3, ISBN-10:1-59059-663-3
Daniel J. Mosley, Bruce A. Posey, “Just Enough Software Test Automation”, Prentice Hall, 2002, ISBN-10: 0130084689, ISBN-13: 9780130084682
Manikandan Sambamurthy, “Test Automation Engineering Handbook”, January 2023, ISBN: 9781804615492
Colin Willcock, Thomas Deiß, Stephan Tobies and Stefan Keil, “An Introduction to TTCN-3” Wiley, 2nd edition 2011, ISBN-10: 0470663065, ISBN-13: 978-0470663066

Articles

Robert V. Binder, Suzanne Miller, “Five Keys to Effective Agile Test Automation for Government Programs” August 24, 2017, Software Engineering Institute, Carnegie Mellon University, https://resources.sei.cmu.edu/asset_files/Webinar/2017_018_101_503516.pdf
DoD CIO, Modern Software Practices “DevSecOps Fundamentals Guidebook: Activities & Tools”, Version 2.2, May 2023, https://dodcio.defense.gov/Portals/0/Documents/Library/DevSecOpsActivitesToolsGuidebookTables.pdf?ver=_Sylg1WJB9K0Jxb2XTvzDQ%3d%3d
Péter Földházi, “Tri-Layer Testing Architecture”, https://www.pnsqc.org/docs/PROP53522057-FoldhaziDraftFinal.pdf
Thomas Pestak, William Rowell, PhD, “Automated Software Testing Practices and Pitfalls”, September 2018, https://www.afit.edu/stat/statcoe_files/Automated%20Software%20Testing%20Practices%20and%20Pitfalls%20Rev%201.pdf
Andrew Pollner, Jim Simpson, Jim Wisnowski, “Automated Software Testing Implementation Guide for Managers and Practitioners”, October 2018, https://www.afit.edu/stat/statcoe_files/0214simp%202%20AST%20IG%20for%20Managers%20and%20Practitioners.pdf
The Selenium Project, “Page object models”, https://www.selenium.dev/documentation/test_practices/encouraged/page_object_models/

10 Appendix A – Learning Objectives/Cognitive Level of Knowledge

The following learning objectives are defined as applying to this syllabus. Each topic in the syllabus will be examined according to the learning objective for it.

The learning objectives begin with an action verb corresponding to its cognitive level of knowledge as listed below.

Level 2: Understand (K2)

The candidate can select the reasons or explanations for statements related to the topic, and can summarize, compare, classify, and give examples for the testing concept.

Action verbs: Classify, compare, differentiate, distinguish, explain, give examples, interpret, summarize

Examples	Notes
Classify test tools according to their purpose and the test activities they support.	
Compare the different test levels.	Can be used to look for similarities, differences, or both.
Differentiate testing from debugging.	Looks for differences between concepts.
Distinguish between project and product risks.	Allows two (or more) concepts to be separately classified.
Explain the impact of context on the test process.	
Give examples of why testing is necessary.	
Infer the root cause of defects from a given profile of failures.	
Summarize the activities of the work product review process.	

Level 3: Apply (K3)

The candidate can carry out a procedure when confronted with a familiar task or select the correct procedure and apply it to a given context.

Action verbs: Apply, implement, prepare, use

Examples	Notes
Apply boundary value analysis to derive test cases from given requirements.	Should refer to a procedure / technique / process etc.
Implement metrics collection methods to support technical and management requirements.	
Prepare installability tests for mobile apps.	
Use traceability to monitor test progress for completeness and consistency with the test objectives, test strategy, and test plan.	Could be used in a LO that wants the candidate to be able to use a technique or procedure. Similar to 'apply'.

Level 4: Analyze (K4)

The candidate can separate information related to a procedure or technique into its constituent parts for better understanding and can distinguish between facts and inferences. A typical application is to analyze a document, software or project situation and propose appropriate actions to solve a problem or task.

Action verbs: Analyze, deconstruct, outline, prioritize, select.

Examples	Notes
Analyze a given project situation to determine which black-box or experience-based test techniques should be applied to achieve specific goals.	Examinable only in combination with a measurable goal of the analysis. Should be of form 'Analyze xxxx to xxxx' (or similar).
Prioritize test cases in a given test suite for execution based on the related product risks.	
Select the appropriate test levels and test types to verify a given set of requirements.	Needed where the selection requires analysis.

Reference

(For the cognitive levels of learning objectives)

Anderson, L. W. and Krathwohl, D. R. (eds) (2001) A Taxonomy for Learning, Teaching, and

Assessing: A Revision of Bloom's Taxonomy of Educational Objectives, Allyn & Bacon

11 Appendix B – Business Outcomes traceability matrix with Learning Objectives

This section lists the traceability between the Business Outcomes and the Learning Objections of Test Automation Engineering Specialist.

Business Outcomes Test Automation Engineering Specialist			B01	B02	B03	B04	B05	B06	B07	B08	B09	B10	B11	B12
Chapter 1	Introduction and objectives for test automation													
1.1	Describe purpose of test automation													
1.1.1	Explain the advantages and disadvantages of test automation	2	X											
1.2	Test automation in the software development lifecycle													
1.2.1	Explain how test automation is applied across different software development lifecycle models	2		X										
1.2.2	Select suitable test automation tools for a given system under test	2		X										
Chapter 2	Preparing for test automation													
2.1	Configuration needs of an infrastructure and development environments													
2.1.1	Describe the configuration needs of an infrastructure that enable implementation of test automation	2			X									
2.1.2	Explain how test automation is leveraged within different environments	2			X									
2.2	Evaluation process for selecting the right tools and strategies													
2.2.1	Analyze a system under test to determine the appropriate test automation solution	4				X								
2.2.2	Illustrate the technical findings of a tool evaluation	4				X								

Business Outcomes Test Automation Engineering Specialist			B01	B02	B03	B04	B05	B06	B07	B08	B09	B10	B11	B12
Chapter 3	Test automation architecture													
3.1	Design concepts leveraged in test automation													
3.1.1	Explain the major capabilities in a test automation architecture	2					X							
3.1.2	Explain how to design a test automation solution	2					X							
3.1.3	Apply layering of test automation frameworks	3					X							
3.1.4	Apply different approaches for automating test cases	3					X							
3.1.5	Apply design principles and design patterns in test automation	3					X							
Chapter 4	Implementing test automation													
4.1	Test automation development													
4.1.1	Apply guidelines that support effective test automation pilot and deployment activities	3						X						
4.2	Risks associated with test automation development													
4.2.1	Analyze Deployment Risks and Plan Mitigation Strategies for Test Automation	4							X					
4.3	Test automation solution maintainability													
4.3.1	Explain which factors support and affect test automation solution maintainability	2								X				
Chapter 5	Implementation and deployment strategies for test automation													
5.1	Integration to CI/CD pipelines													
5.1.1	Apply test automation at different test levels within pipelines	3									X			
5.1.2	Explain configuration management for testware	2									X			

Business Outcomes Test Automation Engineering Specialist			B01	B02	B03	B04	B05	B06	B07	T08	B09	B10	B11	B12
5.1.3	Explain Test Automation Dependencies for an API Infrastructure	2									X			
Chapter 6	Test automation reporting and metrics													
6.1	Collection, analysis and reporting of test automation data													
6.1.1	Apply data collection methods from the test automation solution and the system	3										X		
6.1.2	Analyze data from the test automation solution and the system under test to better understand test results	4										X		
6.1.3	Explain how a test progress report is constructed and published	2										X		
Chapter 7	Verifying the test automation solution													
7.1	Verification of the test automation infrastructure													
7.1.1	Plan to verify the Test Automation Environment Including Test Tool Setup	3											X	
7.1.2	Explain the correct behavior for a given automated test script and/or test suite	2											X	
7.1.3	Identify Where Test Automation Produces Unexpected Results	2											X	
7.1.4	Explain how static analysis can aid test automation code quality	2											X	
Chapter 8	Continuous improvement													
8.1	Continuous improvement opportunities for test automation													
8.1.1	Discover opportunities for improving test cases through data collection and analysis	3												X

Business Outcomes Test Automation Engineering Specialist			B01	B02	B03	B04	B05	B06	B07	B08	B09	B10	B11	B12
8.1.2	Analyze the technical aspects of a deployed test automation solution and provide recommendations for improvement	4												X
8.1.3	Restructure the Automated Testware to Align with SUT Updates	3												X
8.1.4	Summarize opportunities for use of test automation tools	2												X

12 Appendix C – Release Notes

ISTQB® Test Automation Engineering Syllabus 2024 is a major update and rewrite of release 2016. For this reason, there are no detailed release notes per chapter and section. The content has been significantly enhanced for the test automation engineering role, while strategic content has been moved to a new ISTQB® Test Automation Strategy Syllabus 2024.

13 Appendix D – Domain Specific Terms

Term Name	Definition
DevSecOps	A methodology that combines development, security, and operations and uses a high degree of automation.
flow model pattern	A high-level view of the work domain, its components, and interconnections among them.
user journey	A series of steps that show, from the perspective of the person's experience, how a user interact with a system under test.
page object pattern	A design pattern in test automation for enhancing test maintenance and reducing code duplication.
version control	A process to check in and store specific versions of the source code.

14 Index

All terms are defined in the ISTQB® Glossary (http://glossary.istqb.org/).