3 Generic Test Automation Architecture

Test automation is not an end in itself, and should provide valuable information about the quality of the test object, thus enabling the organization and the team to make informed decisions at any point in time. This applies during the development of the test object and throughout the entire product lifecycle. Consequently, test automation is not only useful for testing during development, but also for maintenance testing. However, this also means that test automation itself must be maintained throughout the entire lifecycle while remaining cost effective. This is one of the greatest challenges a test automation engineer must face.

3.1 Introducing Generic Test Automation Architecture (gTAA)

This chapter describes a generic test automation architecture (gTAA) as defined in the ISTQB® *Certified Tester Test Automation Engineer* syllabus. The structure of this chapter is deliberately based on the corresponding chapter in the syllabus. It elaborates on the content therein, places it in a broader context, and supplements it with specific elements, such as a selection of relevant tools, case studies and, finally, an end-to-end example.

The gTAA specifies the basic layers, components, and interfaces that make up a typical test automation solution (TAS). It is intended as a basis for deriving a concrete test automation architecture (TAA) in a structured, modular, and reliable manner for a specific context.

The second half of the chapter addresses how the gTAA can be translated into a specific test automation architecture (TAA) and a test automation solution (TAS) based on that architecture.

3.1.1 Why is a Sustainable Test Automation Architecture important?

As already mentioned in previous sections, the main driver for the beneficial use of test automation is the cost of maintenance. Maintenance costs can be reduced using careful test case selection, and by automating the smallest possible set of meaningful test cases. Additionally, a sustainable and modular test automation architecture (TAA) also supports the long-term economic viability of a test automation solution. Not only does such an architecture produce benefits during the maintenance phase, it also allows faster automated test case development, increased automation stability, and flexibility in the face of changes to the technology stack or the test object.

A modular architecture can also ensure reusability not only within a single team or test object, but also within the wider organization.

Not only the entire architecture, but also specific test automation elements (such as interface connections and tool configurations) can be reused across teams or organizations. This means that, alongside technology connectors, teams can also use the automated test case building blocks that were created with them. This can be especially useful when test cases rely on data or actions that take place in other systems, or when building an automated system integration test suite.

Meeting these requirements makes heavy demands on the TAE. In addition to testing and automation skills, the TAE should also have in-depth knowledge of the chosen development approach, programming standards, best practices, and the domain-specific context.

3.1.2 Developing Test Automation Solutions

A test automation solution (TAS) is a specific instance of a TAA and consists of the test environment and its corresponding automated testware. The latter includes automated test cases (which may be grouped into test suites), test data, and specific configuration files. A TAS is therefore not a monolithic tool or framework, but rather a combination of tools, components, and testware brought together for the purpose of automating testing processes. A test automation framework (TAF) (see section 1.4.4) can, however, be used to provide a test environment, tools, test libraries, or additional test frameworks that can then be reused for faster automated test creation and execution.

There are many factors to consider when developing a TAS. The focus when designing this type of TAS is on:

- Defining the functional and non-functional scope
- Defining the layers, services, and interfaces

- Developing automated test cases efficiently and effectively, using the simplest possible components
- Reusing elements for different technologies, tools, and test objects (for example, product lines)
- Simplifying maintenance and further development
- Fulfilling any other user requirements

This list also shows why the development of a test automation solution is a form of specialized software development: requirements, technical design and interfaces must be identified, specified, implemented, and of course tested. A test automation project is a software development project! It is therefore not surprising that the fundamental principles of software development, the five SOLID principles according to Uncle Bob [Martin 00], must also be applied.

Authors' note: Since there is more than enough excellent material on the SOLID principles, both in print and online, we will not provide a detailed description and code examples here. However, we will provide an overview and examples of their relevance in the context of a TAS. The SOLID principles are not explicitly identified as such in the syllabus, but the titles of the following subsections make it clear which sections are being referred to.

S: The Single-Responsibility Principle

This principle dictates that each component fulfills one (and only one) clearly defined task, which is fully implemented and encapsulated in the component. In the context of a TAS, this can be the generation of specific test data, the execution of tests, the implementation of a specific business task, the connection to an interface, the logging of test actions, or the generation of a report.

A proven method for scoping a task is to think about the question: Why might this component require change?

As an example, let's take a component that is responsible for generating a visually appealing and comprehensive report. What reasons could there be to change it? Perhaps the contents of the report need to be changed—for example, by adding a new field "Duration of test case". Another reason might be a change in the layout—for example, altering the position of diagrams on the page. The fact that we have already identified two possible reasons for change indicates that the "Single-responsibility principle" has been violated. Therefore, in our example, content provision needs to be separated from formatting by splitting the functionality between two separate components.

O: The Open-Closed Principle

Each component should be open to enhancements but closed for modifications. The goal is the ability to add functionality without affecting existing functionality, which is an important factor in maintainability and backward compatibility (i.e., the ability to support previous usage scenarios following a change).

For example, a component is “open” if there is a way to add data or functionality to it. In an object-oriented context this can happen through polymorphism in several ways—for example, through interface definitions or inheritance. However, because the component shows no change externally, in both cases it is still “closed”. In the case of an interface, the signature remains stable, and, in the case of inheritance, the original class remains unchanged, so existing uses of this class are not affected.

An example in the context of a TAS is the expandability of test interfaces to the SUT: It should be easy to make new areas and functionalities of the SUT accessible to new tests without negatively affecting existing ones.

L: The Liskov Substitution Principle

Each component should be replaceable without affecting the overall behavior of the TAS. This means that a component in use can be replaced by another component defined on the same basis (for example, a base class or interface) without affecting the system’s ability to produce the desired result. This implies that the usage of these components is compatible—in other words, all their features are appropriately implemented and the calling order of both components produces no contradictions.

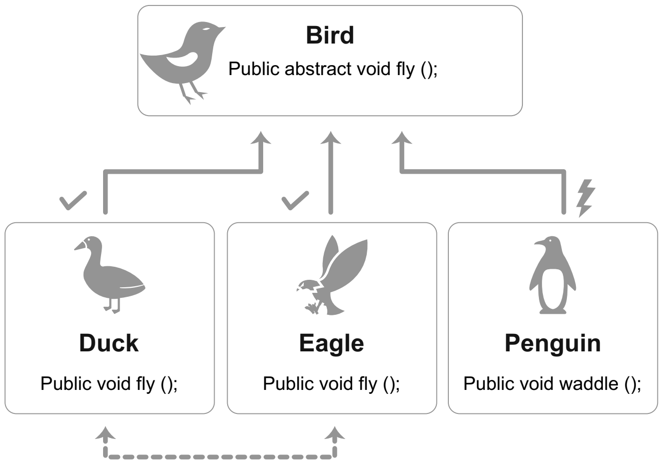


Fig. 3–1
Illustration of the Liskov Substitution Principle

A typical analogy is the “duck” example shown in the illustration. If a system requires a “bird” component with the ability to fly, implementing “duck” can fulfill this role. In this case, “duck” can be replaced by other birds but not, for example, using “penguin”, which violates the substitutability for “duck” because the “fly” functionality isn’t implemented. When an essential attribute isn’t implemented, you can no longer guarantee that the overall system will still work as intended.

In the context of a TAS, typical examples are the implementation of reporting mechanisms with interchangeable adapters for various types of reports and test management systems, or the control of multiple, interchangeable web browsers via a generic interface.

I: The Interface Segregation Principle

The interface segregation principle states that no component should depend on methods that it doesn’t need. To achieve this, it is helpful to design components in a modular fashion so that they don’t contain too much functionality. This principle is closely related to the single responsibility principle described above. The goal is to keep dependencies clear, concise, and maintainable.

This also relates to how interfaces should be designed in general. Within the scope of a TAS, examples that build on the keyword-driven test automation approach are the implementation of keyword libraries (a test case should not include keywords that are not actually used), or of capabilities in *WebDriver/Selenium,* which combines smaller interfaces that describe individual functionalities and has a dictionary for managing which functionalities are required or available in the implementation. This way, the individual interfaces remain lightweight and consistent, and specific functionality can be added as needed.

D: The Dependency Inversion Principle

[Martin 00] states: Components on a higher architectural level should not depend on components on a lower level. Both components should use the same abstraction, while abstractions shouldn’t depend on implementation details. This leads to the required, forced separation of abstraction and implementation (or logic and technical details). In turn, this leads to looser coupling and increased replaceability.

In the context of a TAS, this principle affects many areas, such as the connection to the SUT, the design of keyword libraries, or the provision of reporting mechanisms. An example of a violation is the use of a specific browser in a test script that tests basic business logic. Launching the same test script with a different browser would require the script to be altered.

Instead, an abstraction should be introduced for the connection to the browser and in the script, which improves both modularity and flexibility.

Real-World Examples:
Tool vs. automation solution

Test automation is often associated with commercial off-the-shelf products. The gTAA can be implemented in many of these products, but it is important to note that the actual implementation of the gTAA is specific to the context of the application and to the artifacts created by the tool. Therefore, using a generic tool alone cannot guarantee a systematically implemented gTAA, and we have to consider the actual context we"re using it in.

For example, a conventional automation tool allows the implementation of test scripts and test cases, as well as the integration of external sources of test data. However, the tool itself cannot ensure that the test cases and test scripts fulfill the structure and principles of the gTAA. Appropriate training for, and a systematic approach by the people entrusted with automation are essential.

The gTAA also provides no information about how many and which tools, plug-ins, interfaces, libraries, and frameworks to use when defining a specific TAA. The gTAA is technologically neutral, vendor-independent, and not bound to a specific domain.

By its very definition, the gTAA is generic (the "g" in gTAA). In other words, it can be implemented using a wide range of tools, technologies, and patterns, and can be used for different domains, test objects, test objectives, and test levels if it is reflected in a specific TAA and implemented in a concrete TAS.

This viewpoint confirms once again that test automation is a form of specialized software development, and that corresponding skills and resources are necessary to provide and maintain both testing and the TAS—with its infrastructure and associated artifacts (such as documentation)—with an adequate level of quality.
Company-specific standards, such as code quality, documentation, or artifact and knowledge management are important in this context too.

3.1.3 The Layers in the gTAA

As previously stated, a TAS should be implemented using the tried and trusted practices and principles of software development. In this context, the gTAA is seen as an abstract architecture that supports the design, operation, and maintainability of a TAS and the automated testware that either explicitly or implicitly underlies most of today's test automation solutions. The gTAA is based on SOLID principles, regardless of whether a structured, object-oriented, or service-oriented development approach is preferred. It represents a vendor-neutral and technology-independent reference architecture for deriving concrete TAAs and developing them into one or more TASs. It defines several horizontal, logical layers that not only represent the flow of certain activities and tasks (for example, test design comes before test execution), but also abstraction levels for the testware. These layers, which build on one another, are illustrated in figure 3–2.

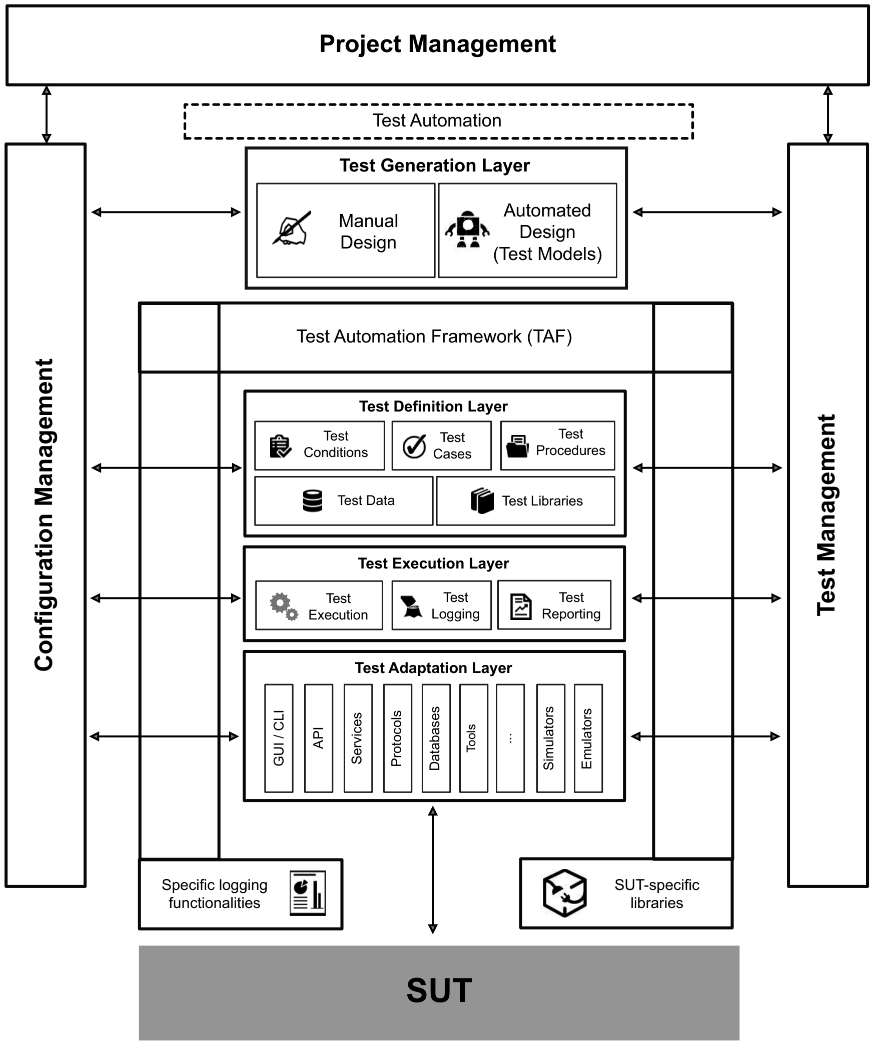


Fig. 3–2
The generic test automation architecture (gTAA)[1] *[ISTQB: CT-TAE]*

Each of these layers has specific tasks, which are summarized below:

- **The test generation layer** supports manual test case design or automated generation of test cases from models that define the SUT and/or its environment

1. Note: The term "test suites" (on the test definition layer) is used in the illustration above in place of the term of "test procedures" that is used in the original illustration in the ISTQB® syllabus. This is because the term "test procedures" is not included in the official ISTQB® glossary and is not explained within the syllabus.

- **The test definition layer** supports the definition and implementation of abstract and concrete test cases and their components (test steps, test data)
- **The test execution layer** supports the execution and logging of tests
- **The test adaptation layer** supports the control of various interfaces that link the TAF to the SUT (for example, GUI, CLI, API; see also Chapter 1)

The gTAA also defines logical interfaces for integration with other tools related to test automation:

- **Project management** controls TAS development as a software project and its integration into the software lifecycle, as well as integration and synchronization with the SUT's project management
- **Configuration management** manages the configurations and versions of all relevant test tools and components in the TAS
- **Test management** provides test protocols, test results, and traceability, and enables test progress evaluation by the test manager

Since the layers of the gTAA are modular, individual layers can be built in various ways, depending on the environment. Certain layers may not be explicitly implemented in a TAS, and functionalities that belong to two layers are not always strictly separated. Furthermore, there may be additional levels of abstraction within each layer (for example, multi-level keywords where higher-level keywords are implemented by lower-level keywords, test data abstraction, and so on). Each layer can also potentially implement integration capabilities for external systems, often necessitating technology- or tool-specific code.

The generic character of this layered model enables us to build different TASs for a wide variety of use cases. This proven model is straightforward and in common use, especially for workflow-based functional testing. However, some use cases—load and performance testing, testing with large data sets, high-level parallelization, real-time requirements, machine learning, probabilistic systems, and so on—may require the model to be extended or modified. If the gTAA itself requires modification, we recommended that you analyze the gTAA and make changes to it deliberately and carefully.

Case Study:
Testing a Data Warehouse System as a Gray Box

This case study uses an example to illustrate how the layers of the gTAA can interact, and how a TAS interacts with its environment. It also shows that the gTAA can form a basis for building a TAS, even in non-workflow-based test setups.

A highly automated testing approach was used for the group data warehouse of a large multinational bank. Part of this approach involved testing a large number of transformation rules for the massive data volume generated by each member of the banking group and their different core banking systems.
From the viewpoint of the layered gTAA model, the TAA underlying the developed TAS was structured as follows:

- **Test generation**
  Automated test cases were generated from a rule model derived semi-automatically from semi-formal business transformation rules.
- **Test definition**
  Test generation produced abstract test cases, which implemented rules in tabular form for testing the results of the individual test data transformations in a programmatically executable manner. Concrete test cases were not available within this layer because the test data was only provided in an ad-hoc manner during execution.
- **Test execution**
  The test execution layer took test data defined and provided in the test definition layer, controlled the loading of the database, executed the transformation program provided by the development team and checked the results against the expected result rules. Discrepancies were logged and the test results, associated log files, and test protocols were imported into the company's test case management system.
- **Test adaptation**
  The relevant interfaces were accessed via adapters to load the database and the transformation programs, and to check the results. Various adapters were used on this layer due to the different systems used by different members of the banking group.

Further gTAA components were also implemented to embed testing as seamlessly as possible in the development process:

- **Project management**
  The implementation of the TAS was embedded within the company as a separate software development project in the DW testing domain. This included planning, requirements gathering, implementation and quality assurance as well as go-live, maintenance, and enhancement activities. The implementation and refinement were largely carried out using the "water-scrum-fall" agile method. The project management and resource management system were used to plan automation activities, test data procurement, and test execution. Resulting defects were manually entered into a defect management system. A technical connection between the defect management system and the TAS wasn't necessary.

→

- **Configuration management**
  A file-based configuration structure was created for configuring the framework, loading the test data into the correct databases, performing the correct transformations, and validating against the expected results. The company's standardized versioning system was used to manage the TAS code base and the configuration files. As a result, all artifacts that implement each of the layers were integrated into the common configuration management structures and processes. The company's test management tool was used to manage and historize test results and reports.
- **Test management**
  The TAF was connected to the company's test management tool on the test execution layer. The expected results were mapped as test cases, and each time they were executed, a corresponding execution object (including its detailed results) was logged in the test case management section. This made reports, metrics, and an overall view of the test cases and their results available at any time. These reports were the main source of information for test progress reports and served as the basis for release go/no-go decisions within the test management and release process.

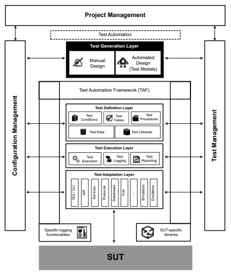
The layer model is often implemented from the bottom up—in other words, starting at the lowest layer and working upward to the uppermost layer. In contrast, top-down implementation (i.e., from the upper, more abstract layers to the lower, more technical layers) is another widely used approach that places increased focus on the writing style of automated test cases.

The Individual Layers in Detail

This section compares the individual layers of the gTAA and describes the differences between them. It also describes their functionalities and supported test activities, and lists concrete examples of tools that can typically be found (or used) within each layer.[2]

2. The tools listed are selected for their symbolic value and easy recognition with respect to the layer in question. They are not intended as specific recommendations and do not represent a universal solution.

Fact Sheet: Test Generation Layer



Tools for:

- Designing test cases
- Defining and managing test data
- Automatically generating test cases

Functionality:

- Traceability to requirements or models
- Modeling and configuration for automatic test case generation

Description:

The test generation layer is the uppermost layer of the gTAA and thus has the highest level of technical abstraction. It is used to capture test case content, test data and test suites, and the traceability of these to other relevant elements of the test basis, such as the requirements or test items that make up the test object.

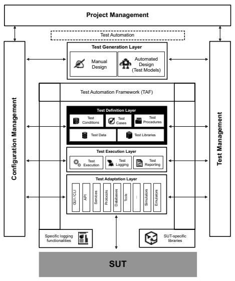
The term “test generation” will of course make many readers think of model-based testing. In this case, a model is derived from requirements, processes, or the SUT, which is itself used as the foundation for deriving test cases according to certain criteria. In many cases, this type of automated test case generation can be associated with the test generation layer. For example, it includes creating the necessary models for defining or configuring the generation algorithms and establishing traceability for the resulting test cases. In some cases, individual test steps are even assigned to the model elements from which they originated.

If manual test case design is facilitated on an abstract level (for example, visually or via text-based, but domain-specific formats or domain-specific languages (DSLs)), this is also considered to be part of the test generation layer. This includes managing such test suites and test cases (i.e., navigating through structures, updating, deleting, and so on), as well as documentation management.

The same applies to the development, collection, or derivation of test data or their technical basis. This includes both manual and automated approaches to test data generation, and linking test data to the underlying requirements or test cases.

Typical tools:

- Gherkin feature file editors
- *Tricentis Tosca TestCase-Design*
- *MBTsuite*



Fact Sheet: Test Definition Layer

Tools for:

- Defining test cases, test conditions, test data
- Specifying test procedures
- Defining test scripts
- Accessing test libraries

Functionality:

- Partitioning, restricting, parameterizing, and instantiating test data
- Specifying, parameterizing, and grouping test sequences and test behavior patterns
- Documenting test cases, test data, test procedures
- Test suite and test case design, test case management and documentation

Description:

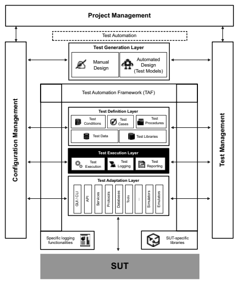
While the test generation layer deals with test case and test data design and management, the test definition layer contains abstract or concrete test cases, test data, test procedures, and the corresponding scripts or code modules (for example, keyword implementations). Tool support for the creation, management, and provision of test suites, test cases, code modules, and scripts is also part of this layer. Here, the focus is not on automatic definition, but rather on the actual implementation and the creation of the corresponding structures. Thus, data-driven or keyword-driven test cases (or "test flows") also belong to this layer.

Components implemented within this layer can support both the implementation and documentation of these elements at a lower level (for example: technical scripts or code modules, concrete manifestations of test data), and the selection and specification of these elements (for example: partitioning, grouping, parameterization, instantiation of test suites, test cases, or test data).

If this takes place after generation but before execution, this can also include the "concretion" or "detailing" of test cases (i.e., the addition of concrete test data to abstract test cases),

Typical tools:

- Gherkin features files & step definitions
- *Tricentis Tosca Modules & TestCases*
- *TestNG* test suites



Fact Sheet: Test Execution Layer

Tools for:

- Automatically executing test cases
- Logging test executions
- Documenting test cases, test data, and test runs

Functionality:

- Setting up, instrumenting, and cleaning up the SUT and test suites
- Configuring and parameterizing the test environment
- Interpreting test data and test cases, and translating them into executable scripts
- Analyzing and validating the SUT's reactions to the tests
- Scheduling test execution

Description:

The test execution layer is in many ways the core of a TAS. It implements the actual execution of the test cases, and thus controls interactions of the TAS with its environment (mainly the SUT).

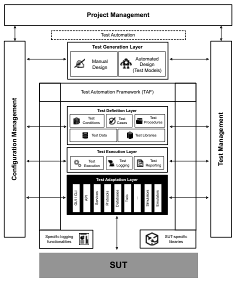
On this layer, the test suites, test cases, and test steps defined in the test definition are interpreted and processed in the specified sequence, and test execution is logged. This layer also implements the parallelization of test suites, test cases, and test steps. Another common practice on this layer is the interpretation (often referred to colloquially as "flattening") of test data or keywords into concrete, executable test steps.

This layer also includes automated checking of preconditions for the execution of automated tests, as well as any corresponding cleanup tasks. This includes setting up and cleaning the SUT and the database, setting TAS parameters and checking the correctness of the test environment based on their configuration. The application of further elements of a test framework (for example, the orchestration of the SUT for technical validations, fault injection, or performance measurements) are also implemented in this layer.

The core goal of a test execution is a test result, either *pass or fail*. This makes it necessary to compare the actual behavior of the SUT with its expected behavior. This comparison (validation) of the SUT's reactions to the test steps is also the responsibility of the test execution layer. Alongside comparison, test execution logging is essential too, especially in the case of failure. Deviations and details of the failed comparison between actual and expected behavior, as well as further processing of an escalation (for example, screenshots, inclusion in a report, or a possible error message) are part of this layer too, as are importing test results into test management tools and generating reports.

Typical tools:

- *JUnit* runner
- *Cucumber* runner
- *Tricentis Tosca TestPlanning & Execution*



Fact Sheet: Test Adaptation Layer

Tools for:

- Controlling the test framework
- Interacting with the SUT
- Monitoring the SUT
- Simulating and emulating (parts of) the SUT's environment

Functionality:

- Accessing the adapter that corresponds to the technology
- Execution of actions based on supported technologies
- If necessary, distribution of test execution among multiple devices

Description:

From a technical point of view, the test adaptation layer is the bottom layer and therefore usually contains many technology-specific elements. In most TASs, the higher layers are implemented in a technology-agnostic way for reasons that include:

- Enabling cross-system or cross-product testing
- Improving long-term maintainability in the face of technology changes
- Swapping out test tools
- Ensuring reusability across systems
- Ensuring reusability across teams and people

Despite all the technical abstractions, it is of course essential for automated testing to interact with the SUT at some point. This interaction takes place via the existing test interfaces and usually involves two typical activities (previously discussed in section 1.5.1):

- Controlling the SUT (for example, clicking a button or invoking a REST service)
- Observing the SUT (for example, reading a text field or receiving a REST response)

In both cases, a technology-specific implementation of the corresponding activity is necessary. The logical or abstract actions are connected to the SUT via an adapter.

Alongside the adaptation of controllability and observability of the SUT, the implementation and control of other elements of the test framework (for example, simulators and emulators, or monitoring and surveillance solutions) are part of the test adaptation layer too.

In the case of parallelization or multi-device testing, tools for the provision and management of appropriate (available) devices for running tests can also be found in this layer.

Typical tools:

- *Selenium*, *Selenium Grid*
- *Appium*, *Ranorex*
- *Xamarin.UITest*
- *BrowserStack*

3.1.4 Project Managing a TAS

Developing a TAS is a software development and rollout project. In addition to simply distributing the TAS, the development, integration, and rollout of the associated processes are also essential aspects and factors that contribute to the success of test automation. As with all software projects, decisions that have to be made for a test automation project include:

- Who are the stakeholders and users?
- What are the goals of the project?
- Which processes are influenced by the solution?
- Which conditions are required or are already in place?
- Which approach will the development project take?
- Which resources are needed?
- Which deliverables are planned?
- How can the TAS be efficiently quality assured?

Test automation often takes place in the context of a software development project, or in an environment in which software development takes place.

Therefore, the answers to many of these questions (especially in procedural and process-based/technical contexts) are usually present in the form of existing elements—for example, a standard procedure, a given technology stack, or existing project management tools.

If this is not the case—for example, if an organization purchases software from a vendor and decides to automate acceptance regression testing—test automation may be one of the first development projects the organization undertakes. This can be challenging and should not be underestimated, and it is important to ensure that there is sufficient focus on, and expertise dedicated to the subject.

Test automation development involves all the same activities that are relevant in a conventional software development lifecycle, so establishing the highly qualified role of the *test automation developer* is a key factor.

Furthermore, the development project should be set up so that important information regarding goals, use of resources, progress, results, and obstacles can be accessed quickly and easily, thus providing plenty of timely opportunities to retain control of the project during its lifetime. Automating the aggregation of this information (for example, using a dashboard) can be very helpful (see Chapter 5).

Real-World Examples:
Illustrating the TAS management process

A TAA is to be developed using an agile approach based on Scrum. The architecture is to be used by the various application teams within the organization, each of which has its own TAEs. Thus, a central TAA will be developed, which will then be used for the implementation of several TASs.

In a series of workshops with stakeholders (for example, the SUT's product owner, management, test automation engineers from the teams), the product owner (in this case the test manager) collects epics for the TAS and describes them in the task-tracking system. These epics are then prioritized in the backlog. The epics in the backlog are broken down into stories and implemented by the TAE. After each sprint, the application teams receive a new version of the TAS that the TAEs use to define and execute regression tests for their particular SUT. Any bugs that occur are also reported to the backlog and prioritized there.

Rule communication and general procedures are applied according to Scrum (estimations, sprint planning, daily Scrum, sprint review, sprint retrospective, board, backlog prioritization, backlog refinement, and so on). This also provides continuous control of focus activities according to the current priorities. A dashboard enables all stakeholders to stay informed about the status of the project. Additionally, the TAA provides deliverable dashboard templates for the regression tests performed by the individual teams.

3.1.5 Configuration Management in a TAS

A TAS is mostly used to test different versions of a SUT. The TAS itself therefore needs to be adapted and modified over time. This aids its evolution while ensuring that it remains compatible with different versions of the SUT. If this compatibility is not ensured, test execution results won't be representative and will, in turn, require a great deal of analysis and maintenance effort.

This makes the configuration management of the TAS in conjunction with the SUT an essential success factor for sustainable test automation.

This affects all aspects of a TAS, including:

- Models
- Test definitions (test data, test cases, libraries)
- Test scripts
- Test execution components
- Test adaptation components
- Simulators and emulators
- Test protocols
- Test reports
- Bug reports

TAA, TAS and automated testware versions must be aligned in order to obtain meaningful results, and each of these elements should be securely versioned to ensure traceability and reproducibility. Knowledge management (for example, for documenting the TAS), task tracking (for activity traceability), a test case management tool (for test results and reports), and code version management (for automation artifacts) are often combined to record and manage all important artifacts in a controlled and versioned fashion.

However, operational management is not the only prerequisite for proactive version management. The communication flow within the project, and between it and its environment, should also be designed in such a way that information on changes to the SUT, the process, or the infrastructure can be actively communicated to the TAEs so that they can act and implement appropriate changes to the TAS. If it isn't, the TAE will end up performing defect analysis for failed test runs, only to discover that planned changes were made somewhere by someone, but were never communicated properly.

3.1.6 Support for Test Management and other Target Groups

Since test automation is a type of software development that takes place within the software testing domain, test management is usually a major stakeholder and, ultimately, the end user. Test automation should therefore provide valuable information for test managers, but also benefit the entire team and the organization.

As a result, the information generated by automated testing should be prepared in a way that is specific to its target group. The design of test case definitions, documentation, and other artifacts should also be structured to meet user requirements. Ultimately, all relevant artifacts should be prepared with their target group in mind. These include:

- Reports
- Logs
- Metrics
- Test suites
- Test definitions
- Documentation
- Information collected about the SUT

This requires the definition of result artifacts and, often, the implementation of integrations—for example, with test case management tools. The results of these are then managed by configuration management.

3.2 Designing a TAA

Following on from the description of the gTAA (see section 3.1) and how a TAS interacts with the surrounding processes (configuration management, project management, test management), this section addresses the question of how a TAA can be designed to suit a specific project situation.

To begin, we will address several fundamental questions on the basics of TAA design. However, the answers to these questions are only valid within the context of a specific project and/or organization. This is why relevant approaches to test case automation (see section 3.2.2), technical considerations regarding the SUT (see section 3.2.3), and issues surrounding the development and quality assurance processes (see section 3.2.4) are then reviewed and discussed in relation to the initial questions.

3.2.1 Fundamental Questions

The design of a specific TAA is based on fundamental questions that, implicitly or explicitly, anticipate certain design decisions and thus guide the design process. Depending on which layer of the gTAA we are looking at, both the underlying considerations and the possible implications can change from case to case.

What Are the Requirements for the TAA?

As with other software development projects, it is important to keep in mind that there are multiple sources of requirements that have different expectations regarding the TAA design, and therefore need to be considered accordingly. For example, do established processes require integration with other systems (such as test management, project management, and others)? Which stakeholder expectations regarding information provided by the TAS—for example, those of the test executor, the test analyst, the test architect, or the test manager—have to be considered and supported by the TAA? Which test types needs to be supported?

For example, a TAA designed for automated component regression testing, which is only used by a team of developers (for unit testing), will look fundamentally different from a TAA designed for automated system integration testing, where stakeholders from different areas (and without development experience) need to understand the test specifications and be able to create new test cases by themselves.

The following sections use a set of concrete requirements defined for a fictional scenario:

- In a complex system landscape with various partially connected subsystems, a system integration test is to be implemented to identify as early as possible any regressions caused by the deployment of new systems
- The relevant subsystems are very diverse in terms of technology, and include modern web applications, client-server architectures, and legacy systems run on host infrastructures
- To increase efficiency, test automation components created during system testing within the individual systems should be reusable for system integration testing
- Detailed defect analysis logs of test results and the corresponding application logs are to be centrally archived and published using the company-wide test management tool
- Creation, ongoing maintenance, and further development are the responsibility of a dedicated team