1 An Introduction to Test Automation and Its Goals

Software development is rapidly becoming an independent area of industrial production. The increasing digitalization of business processes and the increased proliferation of standardized products and services are key drivers for the use of increasingly efficient and effective methods of software testing, such as test automation. The rapid expansion of mobile applications and the constantly changing variety of end-user devices also have a lasting impact.

1.1 Introduction

A key characteristic of the industrialization of society that began at the end of the 18th Century has been the mechanization of energy- and time-consuming manual activities in virtually all production processes. What began more than 200 years ago with the introduction of mechanical looms and steam engines in textile mills in England has become the goal and mantra of all today's manufacturing industries, namely: the continuous increase and optimization of productivity. The aim is always to achieve the desired quantity and quality using the fewest possible resources in the shortest possible time. These resources include human labor, the use of machines and other equipment, and energy.

Software development and software testing on the way to industrial mass production

In the pursuit of continuous improvement and survival in the face of global competition, every industrial company has to constantly optimize its manufacturing processes. The best example of this is the automotive industry, which has repeatedly come up with new ideas and approaches in the areas of process control, production design and measurement, and quality management. The auto industry continues to innovate, influencing other branches of industry too. A look at a car manufacturer's factories and production floor reveals an impressive level of precision in the interaction between man and machine, as well as smooth, highly automated manufacturing processes. A similar pattern can now be seen in many other production processes.

The software development industry is, however, something of a negative exception. Despite many improvements in recent years, it is still a long way from the quality of manufacturing processes found in other industries. This is surprising and perhaps even alarming, as software is the technology that has probably had the greatest impact on social, economic, and technical change in recent decades. This may be because the software industry is still relatively young and hasn't yet reached the maturity of other branches of industry. Perhaps it is because of the intangible nature of software systems, and the technological diversity that makes it so difficult to define and consistently implement standards. Or maybe it is because many still see software development in the context of the liberal, creative arts rather than as an engineering discipline.

Software development has also had to establish itself in the realm of international industrial standards. For example, Revision 4 of the *International Standard Industrial Classification of All Economic Activities* (ISIC), published in August 2008, includes the new section J *Information and Communication*, whereas the previous version hid software development services away at the bottom of the section called *Real estate, renting and business activities* ([ISIC 08], [NACE 08]).

Software development as custom manufacturing

Although the "young industry" argument is losing strength as time goes on, software development is still often seen as an artistic rather than an engineering activity, and is therefore valued differently to the production of thousands of identical door fittings. However, even if software development is not a "real" mass production process, today it can surely be viewed as custom industrial manufacturing.

But what does "industrial" mean in this context? An industrial process is characterized by several features: by the broad application of standards and norms, the intensive use of mechanization, and the fact that it usually involves large quantities and volumes. Viewed using these same attributes, the transformation of software development from an art to a professional discipline is self-evident.

1.1.1 Standards and Norms

Since the inception of software development there have been many and varied attempts to find the ideal development process. Many of these approaches were expedient and represented the state of the art at the time. Rapid technical development, the exponential increase in technical and application-related complexity and constantly growing economic challenges require continuous adaptation of the procedures, languages and process models used in software development—waterfall, V-model, iterative and agile software development; ISO 9001:2008, ISO 15504 (SPICE), CMMI,

ITIL; unstructured, structured, object-oriented programming, ISO/IEC/IEEE 29119 software testing—and that's just the tip of the iceberg. Software testing has also undergone major changes, especially in recent years. Since the establishment of the *International Software Testing Qualifications Board* (ISTQB) in November 2002 and the standardized training it offers for various *Certified Tester* skill levels, the profession and the role of software testers have evolved and are now internationally established [URL: ISTQB]. The ISTQB® training program is continuously expanded and updated and, as of 2021, comprises the following portfolio:

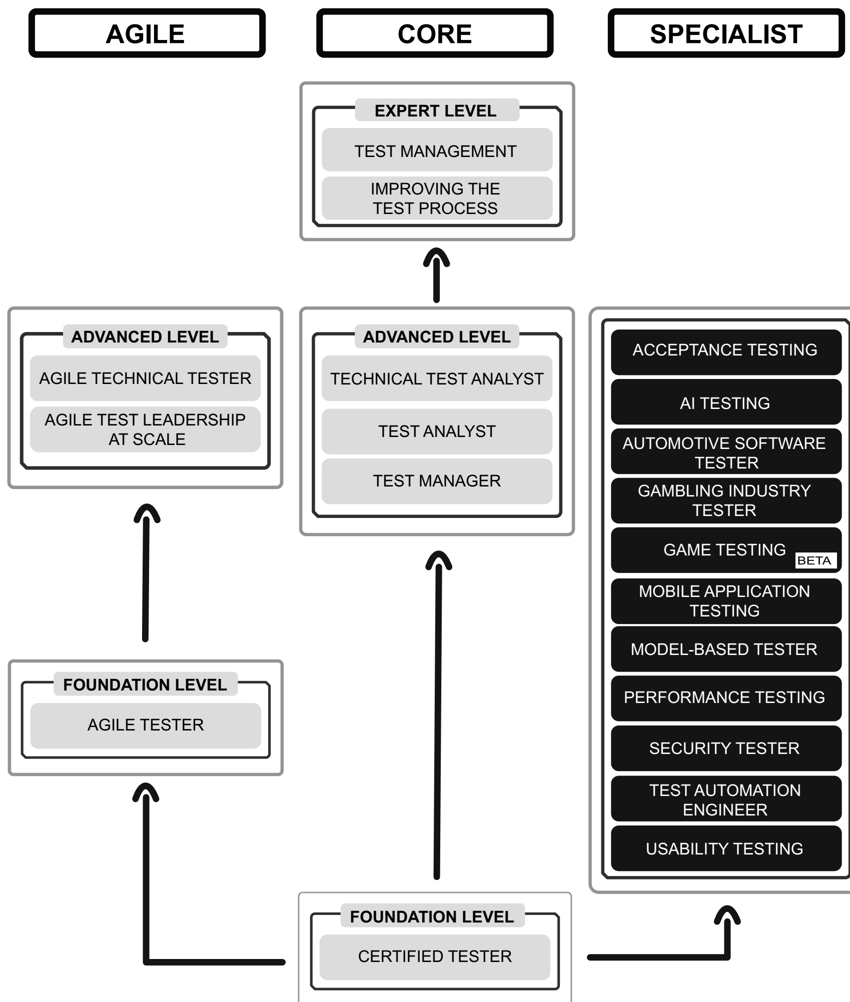


Fig. 1–1
The ISTQB® training product portfolio, as of 2022

Nevertheless, software testing is still in its infancy compared to other engineering disciplines with their hundreds, or even thousands, of years of tradition and development. This relative lack of maturity applies to the subject matter and its pervasiveness in teaching and everyday practice.

One of the main reasons many software projects are still doomed to large-scale failure despite the experience enshrined in its standards is because the best practices involved in software development are largely nonbinding. Anyone ordering software today cannot count on a product made using a verifiable manufacturing standard.

Not only do companies generally decide individually whether to apply certain product and development standards, the perpetuation of the nonbinding nature of standards is often standard practice at many companies too. After all, every project is different. The "Not Invented Here" syndrome remains a constant companion in software development projects [Katz & Allen 1982].

Norms and standards are often missing in test automation

Additionally, in the world of test automation, technical concepts are rarely subject to generalized standards. It is the manufacturers of commercial tools or open source communities who determine the current state of the art. However, these parties are less concerned with creating a generally applicable standard or implementing collective ideas than they are with generating a competitive advantage in the marketplace. After all, standards make tools fundamentally interchangeable—and which company likes to have its market position affected by the creation of standards? One exception to this rule is the *European Telecommunication Standards Institute* (ETSI) [URL: ETSI] testing and test control notation (TTCN-3). In practice, however, the use of this standard is essentially limited to highly specific domains, such as the telecommunications and automotive sectors.

For a company implementing test automation, this usually means committing to a single tool manufacturer. Even in the foreseeable future, it won't be possible to simply transfer a comprehensive, automated test suite from one tool to another, as both the technological concepts and the automation approaches may differ significantly. This also applies to investment in staff training, which also has a strongly tool-related component.

Nevertheless, there are some generally accepted principles in the design, organization, and execution of automated software testing. These factors help to reduce dependency on specific tools and optimize productivity during automation.

The ISTQB® *Certified Tester Advanced Level Test Automation Engineer* course and this book, which includes a wealth of hands-on experience, introduce these fundamental aspects and principles, and provide guidance and recommendations on how to implement a test automation project.

1.1.2 The Use of Machines

Another essential aspect of industrial manufacturing is the use of machines to reduce and replace manual activities. In software development, software itself is such a machine—for example, a development environment that simplifies or enables the creation and management of program code and other software components. However, these "machines" are usually just editing and management systems with certain additional control mechanisms, such as those performed by a compiler. The programs themselves still need to be created by human hands and minds. Programming mechanization is the goal of the model-based approaches, where the tedious work of coding is performed by code generators. The starting point for code generation is a model of the software system in development written, for example, in UML notation. In some areas this technology is already used extensively (for example, in the generation of data access routines) or where specifications are available in formal languages (for example, in the development of embedded systems). On a broad scale, however, software development is still pure craftsmanship.

Mechanization in Software Testing

Use of tools for test case generation and test execution

One task of the software tester is the identification of test conditions and the design of corresponding test cases. Analogous to model-based development approaches, model-based testing (MBT) aims to automatically derive and generate test cases from existing model descriptions of the system under test (SUT). Sample starting points can be object models, use case descriptions or flow graphs written in various notations. By applying a set of semantic rules, domain-oriented test cases are derived based on written specifications. Corresponding parsers also generate abstract test cases from the source code itself, which are then refined into concrete test cases. A variety of suitable test management tools are available for managing these test cases, and such tools can be integrated into different development environments. Like the generation of code from models, the generation of test cases from test models is not yet common practice. One reason for this is that the outcome (i.e., the generated test case) depends to a high degree on the model's quality and the suitability of its description details. In most cases, these factors are not a given.

Another task performed by software testers is the execution and reporting of test cases. At this point, a distinction must be made between tests that are performed on a technical interface level, on system components, and on modules or methods; or functional user-oriented tests that are rather performed via the user interface. For the former, technical tools such as test frameworks, test drivers, unit test frameworks and utility programs are

already in widespread use. These tests are mostly performed by "technicians" who can provide their own "mechanical tools". Functional testing, on the other hand, is largely performed manually by employees from the corresponding business units or by dedicated test analysts. In this area, tools are also available that support and simplify manual test execution, although their usage involves corresponding costs and learning effort. This is one of the reasons why, in the past, the use of test automation tools has not been generally accepted. However, in recent years, further development of these tools has led to a significant improvement in their cost-benefit ratio. The simplification of automated test case creation and maintainability due to the increasing separation of business logic and technical implementation has led to automation providing an initial payoff when complex manual tests are automated for the first time, rather than only when huge numbers of test cases need to be executed or the nth regression test needs to be repeated.

1.1.3 Quantities and Volumes

While programming involves the one-time development of a limited number of programs or objects and methods that, at best, are then adapted or corrected, testing involves a theoretically unlimited number of test cases. In real-world situations, the number of test cases usually runs into hundreds or thousands. A single input form or processing algorithm that has been developed once must be tested countless times using different input and dialog variations or, for a data-driven test, by entering hundreds of contracts using different tariffs. However, these tests aren't created and executed just once. With each change to the system, regression tests have to be performed and adjusted to prove the system's continuing functionality. To detect the potential side effects of changes, each test run should provide the maximum possible test coverage. However, experience has shown that this is not usually feasible due to cost and time constraints.

The required scope of testing can only be effectively handled with the help of mechanization

This requirement for the management of large volumes and quantities screams out for the use of industrial mechanization—i.e., test automation solutions. And, if the situation doesn't scream, the testers do! Unlike machines, testers show human reactions such as frustration, lack of concentration, or impatience when performing the same test case for the tenth time. In such situations, individual prioritization may lead to the wrong, mission-critical test case being dropped.

In view of these factors, it is surprising that test automation hasn't been in universal use since way back. A lack of standardization, unattractive cost-benefit ratios, and the limited capabilities of the available tools may have been reasons for this. Today, however, there is simply no alternative to test automation. Increasing complexity in software systems and the resulting

need for testing, increasing pressure on time and costs, the widespread adoption of agile development approaches, and the rise of mobile applications are forcing companies to rely on ongoing test automation in their software development projects.

1.2 What is Test Automation?

The ISTQB® definition of test automation is: "The use of software to perform or support test activities". You could also say: "Test automation is the execution of otherwise manual test activities by machines". The concept thus includes all activities for testing software quality during the development process, including the various development phases and test levels, and the corresponding activities of the developers, testers, analysts, and users involved in the project.

Accordingly, test automation is not just about executing a test suite, but rather encompasses the entire process of creating and deploying all kinds of testware. In other words, all the work items required to plan, design, execute, evaluate, and report on automated tests.

Relevant testware includes:

- **Software**
  Various tools (automation tools, test frameworks, virtualization solutions, and so on) are required to manage, design, implement, execute, and evaluate automated test suites. The selection and deployment of these tools is a complex task that depends on the technology and scope of the SUT and the selected test automation strategy.

- **Documentation**
  This not only includes the documentation of the test tools in use, but also all available business and technical specifications, and the architecture and the interfaces of the SUT.

- **Test cases**
  Test cases, whether abstract or specific, form the basis for the implementation of automated tests. Their selection, prioritization, and functional quality (for example: functional relevance, functional coverage, accuracy) as well as the quality of their description have a significant influence on the long-term cost-benefit ratio of a test automation solution (TAS) and thus directly on its long-term viability.

- **Test data**
  Test data is the fuel that drives test execution. It is used to control test scenarios and to calculate and verify test results. It provides dynamic

input values, fixed or variable parameters, and (configuration) data on which processing is based. The generation, production, and recovery of existing and process data for and by test automation processes require special attention. Incorrect test data (such as faulty test scripts) lead to incorrect test results and can severely hinder testing progress. On the other hand, test data provides the opportunity to fully leverage the potential of test automation. The importance and complexity of efficient and well-organized test data management is reflected in the GTB *Certified Tester Foundation Level Test Data Specialist* [GTB: TDS] training course (only in German).

- **Test environments**
  Setting up test environments is usually a highly complex task and is naturally dependent on the complexity of the SUT as well as on the technical and organizational environment at the company. It is therefore important to discuss general operation, test environment management, application management, and so on, with all stakeholders in advance. It is essential to clarify who is responsible for providing the SUT, the required third-party systems, the databases, and the test automation solution within the test environment, and for granting the necessary access rights and monitoring execution.

  If possible, the test automation solution should be run separately from the SUT to avoid interference. Embedded systems are an exception because the test software needs to be integrated with the SUT.

Although the term "test automation" refers to all activities involved in the testing process, in practice it is commonly associated with the automated execution of tests using specialized tools or software.

In this process, one or more tasks that are defined the same way as they are for the execution of dynamic tests [Spillner & Linz 21], are executed based on the previously mentioned testware:

- Implement the automated test cases based on the existing specifications, the business test cases and the SUT, and provide them with test data.
- Define and control the preconditions for automated execution.
- Execute, control, and monitor the resulting automated test suites.
- Log and interpret the results of execution—i.e., compare actual to expected results and provide appropriate reports.

From a technical point of view, the implementation of automated tests can take place on different architectural levels. When replacing manual test execution, automation accesses the graphical user interface (GUI testing) or, depending on the type of application, the command line interface of the SUT

(CLI testing). One level deeper, automation can be implemented through the public interfaces of the SUT's classes, modules, and libraries (API testing) and also through corresponding services (service testing) and protocols (protocol testing). Test cases implemented at this lower architectural level have the advantage of being less sensitive to frequent changes in the user interfaces. In addition to being much easier to maintain, this approach usually has a significant performance advantage over GUI-based automation. Valuable tests can be performed before the software is deployed to a runtime environment—for example, unit tests can be used to perform automated testing of individual software components for each build before these components are fully integrated and packaged with the software product. The *test automation pyramid* popularized by Mike Cohn illustrates the targeted distribution of automated tests based on their cost-benefit efficiency over time [Cohn 2009].

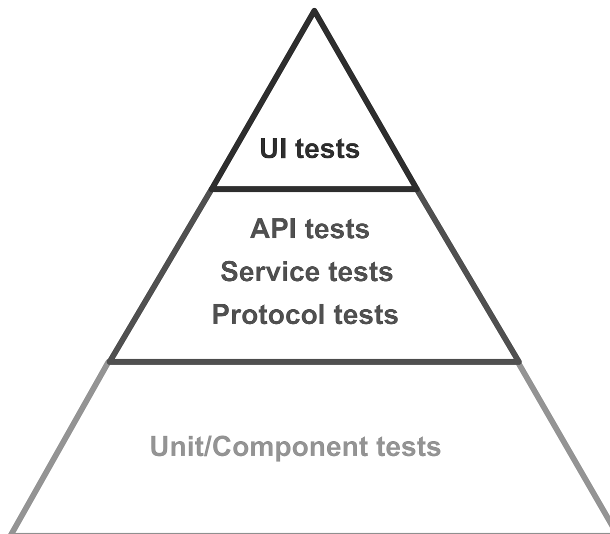


Fig. 1–2
The test automation pyramid

1.3 Test Automation Goals

The implementation of test automation is usually associated with several goals and expectations. In spite of all its benefits, automation is not (and will never be) an end in itself. The initial goal is to improve test efficiency and thus reduce the overall cost of testing. Other important factors are the reduction of test execution time, shorter test cycles, and the resulting chance to increase the frequency of test executions. This is especially important for the DevOps and DevTestOps approaches to testing. Continuous integration,

continuous deployment, and continuous testing can only be effectively implemented using a properly functioning test automation solution.

In addition to reducing costs and speeding up the test execution phase, maintaining or increasing quality is also an important test automation goal. Quality can be achieved by increasing functional coverage and by implementing tests that can only be performed manually using significant investments in time and resources. Examples include testing a very large number of relevant data configurations or variations, testing for fault tolerance (i.e., test execution at the API/service level with faulty input data to evaluate the stability of the SUT), or performance testing in its various forms. Also, the uniform and repeated execution of entire test suites against different versions of the SUT (regression testing) or in different environments (different browsers and versions on a variety of mobile devices) is only economically feasible if the tests involved are automated.

Benefits of Test Automation

One of the greatest benefits of test automation results from building an automated regression test suite that enables increasing numbers of test cases to be executed per software release. Manual regression testing very quickly reaches the limits of feasibility and cost-effectiveness. It also ties up valuable manual resources and becomes less effective with every execution, mainly due to the testers' unavoidable decline in concentration and motivation. In contrast, automated tests run faster, are less susceptible to operational errors and, once they have been created, complex test scenarios can be repeated as often as necessary. Manual test execution requires great effort to understand the increasing complexity of the test sequences involved and to execute them with consistent quality.

Certain types of tests are barely feasible in a manual test environment, while the implementation and execution of distributed and parallel tests is relatively simple to automate—for example, for the execution of load, performance, and stress tests. Real-time tests—for example, in control systems technology—also require appropriate tools.

Since automated test cases and test scenarios are created within a defined framework and (in contrast to manual test cases) are formally described in a uniform way, they do not allow any room for interpretation, and thus increase test consistency and repeatability as well as the overall reliability of the SUT.

From the overall project point of view there are also significant advantages to using test automation. Immediate feedback regarding the quality of the SUT significantly accelerates the project workflow. Existing problems are identified within hours instead of days or weeks and can be fixed before the effort required for correction increases even further.

Test automation also enables more efficient and effective use of testing resources. This applies not only to technical infrastructures, but also to testers in IT and business units, especially through the automation of regression testing. As a result, these testers can devote more time to finding defects—for example, through explorative testing or the targeted use of various dynamic manual testing procedures.

Drawbacks of Test Automation

As well as advantages, test automation has drawbacks too, and these need to be considered in advance to avoid unpleasant surprises later on.

Automating processes always involves additional costs, and test automation is no exception. The initial investments required to set up and launch a test automation solution include tools (for example, for test execution) that have to be purchased or developed; workplace equipment for test automation engineers (TAE) (which usually includes several development and execution PCs/screens); test environment upgrades; the establishment of new processes and work steps that become necessary for developing the test scripts; additional configuration management and versioning systems; and so on.

In addition to investing in additional technologies or processes, time and money need to be invested in expanding the test team's skills. This includes training to become an ISTQB® Test Automation Engineer, further training in software development, and training in the use of the test automation solution and its tools.

The effort required to maintain a test automation solution and its automated testware —first and foremost of course, the test scripts—is also frequently underestimated. Ultimately, test automation itself generates software that needs to be maintained. An unsuitable architecture, noncompliance with conventions, inadequate documentation, and lack of configuration management all have dramatic effects as soon as the automated test suite reaches a level at which changes and enhancements take place constantly. The user interface, processes, technical aspects, and business rules in the SUT change too, and these changes have a direct and immediate impact on the test automation solution and the automated testware.

It is not uncommon for a test automation engineer to find out about such changes "in production" when a discrepancy occurs during test execution. This discrepancy is then reported and rejected by the developer as a defect in the TAS (a so-called "false positive" result). But this is not the only scenario in which the TAS leads to failures—as previously mentioned, a TAS is also just software, and software is always prone to defects.

For this reason, test automation engineers often focus too much on the technical aspects of the TAS and get distracted from the underlying qualitative test objectives that are necessary for the required coverage of the SUT.

Once a TAS is established and working well, testers are tempted to automate everything, such as extensive end-to-end testing, intertwined dialog sequences, or complicated workflows. This sounds like a great thing to do, but you must be aware of the effort involved in implementing and maintaining automated tests. Just creating and maintaining consistent test data across multiple systems for extensive end-to-end testing is a major challenge.

The Limitations of Test Automation

Test automation also has its limits. While the technical options are manifold, sometimes the cost of automating certain manual tests is not proportional to the benefit.

A machine can only check real, machine-interpretable results and to do so requires a “test oracle” which also needs to be automated in some way. The main strength of test automation lies in the precise comparison of expected and actual behavior within the SUT, while its weakness lies in the validation of the system and the evaluation of its suitability for its intended use. Faults in requirement definition or incorrect interpretation of requirements are not detected by the test automation solution. A test automation solution cannot “read between the lines” or apply creativity, and therefore cannot completely replace (manual) structured dynamic testing or exploratory testing. The SUT needs to achieve a certain level of stability and freedom from defects at its user and system interfaces for test sequences to be usefully automated without being subjected to constant changes.

1.4 Success Factors in Test Automation

To achieve the set goals, to meet expectations in the long term, and to keep obstacles to a minimum, the following success factors are of particular importance for ongoing test automation projects. The more these are fulfilled, the greater the probability that the test automation project will be a success. In practice, it is rare that all these criteria are fulfilled, and it is not absolutely necessary that they are. The general project framework and success criteria need to be examined before the project starts and continuously analyzed during the project’s lifetime. Each approach has its own risks in the context of a specific project, and you have to be aware of which success factors are fulfilled and which are not. Accordingly, the test automation strategy and architecture need to be continuously adapted to changing conditions.

Please note: in the following sections we won’t go into any further detail on success factors for piloting test automation projects.

1.4.1 Test Automation Strategy

The test automation strategy is a high-level plan for achieving the long-term goals of test automation under given conditions and constraints. Statements concerning the test automation strategy can be included in a company's testing policy and/or in its organizational test strategy. The latter defines the generic requirements for testing in one or more projects within an organization, including details on how testing should be performed, and is usually aligned with overall testing policy.

Every test automation project requires a pragmatic and consistent test automation strategy that is aligned with the maintainability of the test automation solution and the consistency of the SUT.

Because the SUT itself can consist of various old and new functional and technical areas, and because it includes applications and components run on different platforms, it is likely that specific strategies have to be defined in addition to the existing baseline strategy. The costs, benefits, and risks involved in applying the strategy to the various areas of the SUT must be considered.

Another key requirement of the test automation strategy is to ensure the comparability of test results from automated test cases executed through the SUT's various interfaces (for example, the API and the GUI).

You will gain experience continuously in the course of a project. The SUT will change, and the project goals can be adapted accordingly. Correspondingly, the test strategy needs to be continuously adapted and improved too. Improvement processes and structures therefore have to be defined as part of the strategy.

Excursus: The Test Automation Manifesto

Fundamental principles for test automation in projects or companies can be articulated to serve as a mindset and guide when tackling various issues. The diagram below shows an example from the authors' own project environment:

Test Automation Manifesto

Transparency over Comfort	Collaboration over Independence	Quality over Quantity	Flexibility over Continuity
Test automation must be highly visible to generate added value. We enable transparency, even if this means that we have to expose mistakes in our own work.	It is better to collaborate and connect with other stakeholders and organizations than to solve problems on your own.	Reliable results that drive further work are more important than a high number of automated test cases.	Rather than rigid structures, we prefer a flexible approach that can withstand future challenges.

Fig. 1–3 *The Test Automation Manifesto*

Transparency over **Comfort**

Test automation is characterized by risk calculation and risk avoidance, similar to the safety net used by a high-wire act. This means that if everything works out correctly, regression-testing output (i.e., the number of detected defects) is minimal. However, this doesn't mean that test automation does not add value. It is important to position test automation and its results and functions clearly and visibly within the organization. It also means that any problems with test automation problems are clearly and instantly visible. We believe this to be a strength, not a weakness.

Collaboration over **Independence**

A typical situation occurs when a test automation tool is purchased and handed over to a tester who is then responsible for its implementation and use. Often, the tester in question will enter "experimental mode" and try to implement automated test cases under pressure. A typical behavior pattern in this context is: "Me vs. tool vs. the product"—i.e., a tendency to want to solve or work around problems and challenges alone. Instead, we recommend actively engaging with other roles. For example, if it is difficult to display a particular table, reach out to the developers, ask the community, or simply call vendor support.

Quality over **Quantity**

A typical metric for the value and progress of test automation is the degree of automation of a test suite, measured either as a percentage or the absolute number of automated test cases. However, this does not reflect the additional value generated by the maintainability and robustness of the automated tests. A guiding principle in this context is: "Ten meaningful, stably automated tests are worth more than a thousand unstable and untraceable test cases". Ergo, a small regression test suite is often more useful than a huge test portfolio that is difficult to maintain.

Flexibility over **Continuity**

Test automation is like a twin of the systems it tests and is often a tool for ensuring the successful execution of business processes. It delivers the greatest added value when it can be used over a long period of time with little maintenance. During this time, technologies, tools, personnel, and even business processes can change significantly. To remain effective, test automation requires a high degree of flexibility in the face of change. This is both a strategic and process-related problem as well as a technological/architectural one, which is also addressed by the generic test automation architecture described in detail in later chapters.

A test automation strategy also needs to be tailored to the type of project it is used in. Additionally, the different test levels and test types that are to be supported through automation may also require different approaches.

Section 1.5 on test levels and project types, Appendix A and B provide an introduction to this topic in the form of an excursus (i.e., they are not a part of the official ISTQB® CT-TAE syllabus).

1.4.2 Test Automation Architecture (TAA)

The architecture of a test automation solution is crucial to its acceptance, its existence, and its long-term use. The design of a suitable TAA is also a core topic of the *Test Automation Engineer* training. It requires a certain amount of experience to implement architectural requirements in the best possible way. For this reason, many test automation projects have a test automation architect who, like a software development architect, supports the project in its initial stages and in the case of major modifications.

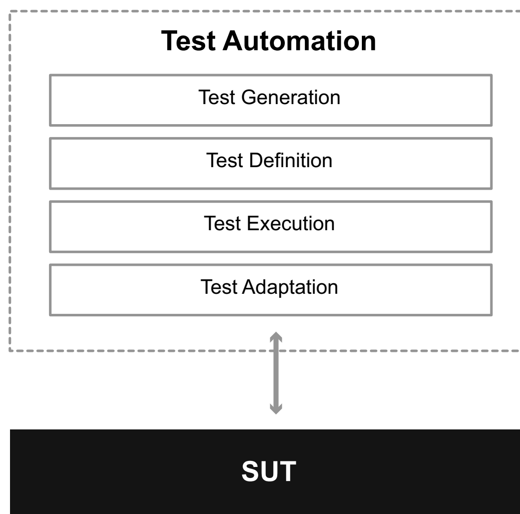


Fig. 1–4
Schematic representation of the layers of a generic test automation architecture (gTAA)

Test Automation Architecture Requirements

- The architecture of a test automation solution is closely related to the **architecture of the SUT**. The individual components, user interfaces, dialogs, interfaces, services and technical concepts, languages used, and so on, must all be addressed.
- The test and test automation strategy should clearly define which **functional and non-functional requirements of the SUT** are to be addressed and supported by test automation, and thus by the test automation architecture. These will usually be the most important requirements for the software product. Appendix A provides an overview of software quality characteristics according to ISO 25010 (part of the ISO/IEC 25000:2014 series of standards).
- However, **the functional and non-functional requirements of a test automation solution** also have to be considered. In particular, the requirements covering maintainability, performance, and learnability are in focus during the design of a test automation architecture. The SUT is

subject to continuous development, so a high degree of modifiability and extensibility is essential. Using modular concepts or separating functional and technical implementation layers are ways to ensure this.

As the size of the automated test suite increases, the performance of the test automation solution becomes an important issue. Increased testing via the API interfaces rather than the GUI can lead to significant improvements in efficiency. Additionally, the test automation solution should not be treated as a mystery that is only accessible to a chosen few experts. Understandability and learnability are therefore also important factors. It is also worth looking at the quality characteristics listed in Appendix A and evaluating them for their potential use within the test automation architecture.

Collaboration with the software developers and architects is essential to develop the best possible architecture for a test automation solution in a given context. This is because a deep understanding of the SUT architecture is required to meet the requirements mentioned above.

1.4.3 Testability of the SUT

Testability or, more precisely, the automated testability of the SUT, is also a key success factor. The test automation tools must have access to the objects and elements of the various user and system interfaces, as well as to system architecture components and services, to identify and leverage them.

Test automation tools provide a range of automation adapters based on a wide variety of technologies and platforms. Whether .NET, Java, SAP, web, desktop or mobile solution, Windows, Linux, Android/iOS, Google Chrome, Internet Explorer, Microsoft Edge, Mozilla Firefox, or Safari, the range is huge.

Manufacturers align their solutions with the common standards used by these technologies and platforms. Problems often arise when the SUT contains implementations and concepts that deviate from these standards. It is therefore necessary to determine the basic automation capability of the SUT during a proof of concept, and to find the most suitable automation solution. Three aspects of this process can be tricky and/or expensive: persuading the manufacturer of an automation tool to modify their product to fit your ideas; convincing the development department to adapt the architecture of the SUT and exchange in-house class libraries for others; somehow finding a workaround using complex constructs within the test automation solution.

However, as the use of test automation becomes more widespread, especially in agile development scenarios, the ability to automate test execution may gain importance as a new quality metric for software applications.

For example, for automated testing via the GUI, the interaction elements and data should be decoupled from their layout as far as possible. For API testing, corresponding interfaces (classes, modules/components, or the command line) can be provided publicly or developed separately.

For each SUT there are areas (classes, modules, functional units) that are easy to automate and areas where automation can be very time-consuming. Potential showstoppers should already have been addressed during tool evaluation and selection. Because an important success factor is the easiest possible implementation and distribution of automated test scripts, the initial focus should be on test areas that can be easily automated. The proof of successful automated test execution helps the project along and supports investment in the expansion of test execution. However, if you dive too deep into critical areas, you may not deliver many results and thus add less value to the project.

1.4.4 Test Automation Framework

A test automation framework (TAF) must be easy to use, well documented and, above all, easy to maintain. The foundations for these attributes are laid in the test automation architecture. The test automation framework should also ensure a consistent approach to test automation.

The following factors are especially important:

- **Implementing reporting facilities**
  Test reports provide information about the quality of the SUT (passed/failed/faulty/not executed/aborted, statistical data, and so on) and should present this information in an appropriate format for the various stakeholders (testers, test managers, developers, project managers, and other stakeholders).
- **Support for easy troubleshooting and debugging**
  In addition to test execution and logging, a test automation framework should provide an easy way to troubleshoot failed tests. The following are some of the reasons for failures and, ideally, the framework will classify them in a way that supports failure analysis:
  - Failures found in the SUT
  - Failures found in the test automation solution (TAS)
  - Problems with the tests themselves (for example, flawed test cases
  - Problems with the test environment (for example, non-functioning services, missing test data, and so on)
- **Correct setup of the test environment**
  Automated test execution requires a dedicated test environment that integrates the various test tools in a consistent manner. If the automated

test environment or the test data cannot be manipulated or configured, the test scripts might not be set up and executed according to the test execution requirements. This in turn may lead to unreliable, misleading, or even incorrect test results (false positive or false negative results). A false positive test result means that a problem is detected (i.e., the automated test fails), even if there is no defect in the SUT. A false negative test result indicates that a test is successful (i.e., the automated test does not encounter a failure), even though the system is faulty.

- **Documentation of automated test cases**
  The goals of test automation must be clearly defined and described. Which parts of the software should be tested and to what extent? Which test automation approach should be used? Which (functional and non-functional) properties of the SUT are to be automatically tested? Furthermore, the documentation of the automated test cases (or test case sets) must make it clear which test objective they cover.
- **Traceability of automated testing**
  The functional test scenarios covered by automated test suites are sometimes exceedingly hard to understand, let alone discover. This frequently results in the creation and implementation of new, redundant test scripts. In addition to a fundamental lack of transparency, this creates a lot of unnecessary redundancies and a lack of clarity. Therefore, the test automation framework must also support traceability between the automated test case steps and the corresponding functional test cases and test scenarios.
- **High maintainability**
  One of the biggest risks for the success of a test automation project is the maintenance effort it involves. Ideally, the effort required to maintain existing test scripts should be a small percentage of the overall test automation effort. In addition, the effort required to customize the test automation solution should be in a healthy proportion to the scope of the changes to the SUT. If test automation becomes more expensive than the development of the SUT, the goal of reducing costs using test automation will probably not be achieved. Automated test cases should therefore be easy to analyze, change, and extend. A good modular design tailored to the SUT allows a high degree of reusability for individual components and thus reduces the number of artifacts that have to be adapted when changes become necessary.
- **Keeping test cases up to date**
  Some test cases fail because changes are made to the business or technical requirements that are not yet addressed in the test scripts, rather than due to an application defect. The affected test cases should not simply be

discarded but rather adapted accordingly. It is therefore essential that the test automation engineer receives all relevant information about changes to the SUT through appropriate processes, documentation, and tools, and can thus update the test suite in a timely fashion.

- **Software deployment planning**
  The test automation framework should also support the version and configuration management built into the test automation solution, which in turn needs to be kept in sync with the current version of the SUT, again through the appropriate use of tools and standards. The deployment, modification, and redeployment of test scripts must be kept as simple as possible.
- **Retiring automated tests**
  When certain automated test sequences are no longer needed, the test automation framework needs to support their structured removal from the test suite. In most cases, it is not sufficient to simply delete scripts. To maintain the consistency of the test automation solution, all dependencies between the components involved must be easy to edit and resolve. As you do when developing software, you should always avoid producing dead code.
- **SUT monitoring and recovery**
  Normally, to be able to continuously execute tests, the SUT needs to be constantly monitored. If a fatal failure occurs in the SUT (a crash, for example), the test automation framework must be able to skip the current test case, return the SUT to a consistent state, and proceed with the execution of the next test case.

Maintaining Test Automation Code

Test automation code can be just as extensive as development code and can also be quite complex. This is certainly the case if intricate or complicated test sequences are implemented within a test script, or if technical interfaces or user interface elements have to be handled in a specific way. You may also have to implement time-based triggers or delays, or chain test steps that are linked to each other via (intermediate) results data. This makes the corresponding maintenance complex, and the effort required increases accordingly. Additionally, there are often multiple test tools in use, different types of verification and validation, and diverse test resources that have to be maintained (for example, test input data, test oracles, and test reports). As for the test automation architecture, maintainability is of the utmost importance for test code and test scripts too.

Recommendations for Reducing Maintenance Effort:

- **Technical independence**
  It is important to avoid (or at least minimize) technical dependencies and links to the SUT. For this reason, the various frameworks and automation tools relocate specific technical links to the GUI and API interfaces on a separate, central layer. The separation of technical and functional aspects is essential, and a test automation engineer should never neglect this aspect of the work.
- **Data independence**
  What applies to the technical links to the SUT also applies to the corresponding test automation base, transaction, and control data, which need to be abstracted into a separate data access layer. Hardcoded test data in the test scripts should be avoided—for example, during test verification. Data changes, such as a new tax rate or a changed confirmation message should not result in the rewriting of numerous scripts. You also have to consider the risks involved in making changes where dependencies exist between test scripts via their input and output data.
- **Environmental independence**
  The implemented set of automated test cases should also be executable in multiple test environments and on multiple platforms. Automation settings, data taken from the operating platform (such as system time or OS localization parameters), or data from other applications within the test environment should be implemented using placeholders or configuration files and settings. Many of these aspects should be provided and used by the test automation framework.
- **Documentation**
  Good (inline) documentation is a great aid to test script modification and extension as well as to debugging. Development and documentation conventions that improve readability and comprehensibility significantly reduce the overall maintenance effort.

1.5 Excursus: Test Levels and Project Types

The definition of a test automation strategy, the design of a test automation architecture, and the development of a test automation framework all take place within a specific context. The automation of test activities takes place during different phases of the development process and on different test levels and—depending on the project type—the strategic, methodological, and technical approaches to test automation may also

→

vary. The following sections provide tips and ideas for designing a suitable strategy, architecture, and framework.

1.5.1 Test Automation on Different Test Levels

There are many models on which software development processes can be based. One widely-used model is the V-model, which provides a basis for the classification of activities and their dependencies. During testing, the various test levels are based on this model, while automation plays a different role depending on the test level concerned.

Today, the V-Model has less of a real-life presence as a real-life model for software development processes, but its phases and levels have become common terms.

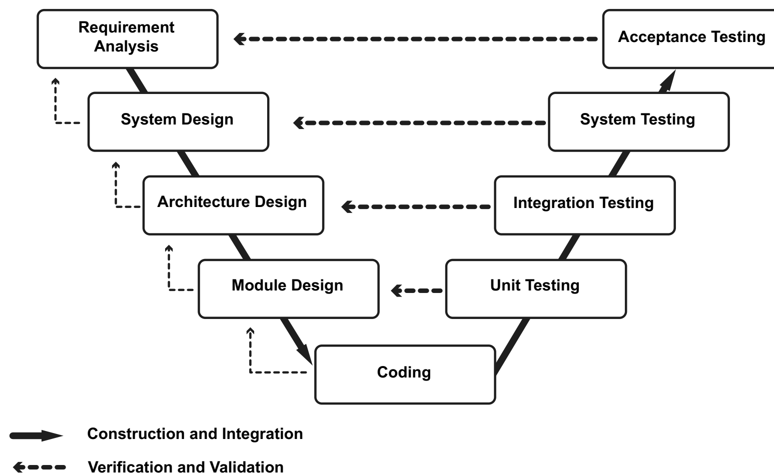


Fig. 1–5
The V-model

Unit Testing

Typical examples of automated tests are unit, module, and component tests. These are usually the responsibility of the development team and are therefore often referred to as “automated developer tests”.

Unit testing verifies functionality within the smallest units of software that can be tested in a meaningful way. Common definitions for such units are classes, functions, methods, or procedures, although other definitions are possible, depending on the language paradigm you are using.

→

Since the SUT usually consists of interdependent units, a testing framework must be created in which these units can be executed and controlled in isolation.

Mocks

In object-oriented environments, this is done by replacing dependencies with simple "mocks" that—unlike the dependencies they represent—have as little functionality as possible. Mocks usually allow you to specify which values they return on calls and to check whether or how often methods have been called.

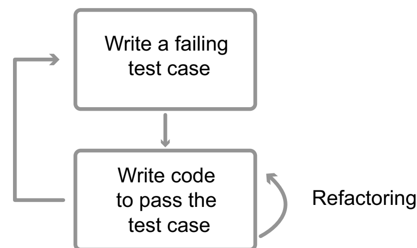
This makes the smallest possible test objects testable in isolation. Automated unit testing is particularly suitable for providing the development team with rapid feedback on the effects of changes to the test object, and thus provides continual security regarding potential changes made to existing functionality by major changes within a unit (for example, refactoring).

The robustness of individual components can also be effectively tested at this test level since individual components can usually be accessed without the restrictions made by upstream data validations.

Test-driven development

Unit testing is increasingly used in the context of test-driven development (TDD). TDD is a central component of many agile software development methods, such as Extreme Programming (XP). The idea behind TDD is that writing code is driven by testing. Typically, tests are written in parallel with, or following, code implementation. The downside of this approach is that high test coverage through automated tests is only possible with a great deal of effort.

Fig. 1–6
The basic approach to test-driven development



TDD takes a different approach that requires a radical shift in mindset, namely: test first, code second. Only as much new code is written as is required for the automated test to be executed so that no defects are reported. The code must be as simple and comprehensible as possible. The advantage is obvious: at any point in the development cycle, there is a set of automated tests that verifies the current code in its entirety.

→

The following steps need to be considered when using this method:

1. Creation of the required (SUT) class
2. Creation of a test class for the (SUT) class
3. Definition and creation of the methods within the (SUT) class and the test class
4. Implementation of the test cases within the methods of the test class
   a) Definition of the input data
   b) Definition of the expected results
   c) Using assertions to check for correctness and failures (worst case)
5. Implementation of logic within the methods of the (SUT) class

Tool support (frameworks) at the test development, test automation, and build automation levels are essential for the successful use of TDD.

Integration Testing

Integration testing describes the explicit testing of the interaction of multiple units or components. Depending on the situation, both unit and system testing methods can be applied on this test level.

- **Unit integration testing**
  Unit integration testing doesn't isolate the unit under test using techniques such as mocking (as used in unit testing), but instead tests a component's interaction with any corresponding components. This type of testing is usually fully automated and offers a good level of security during broad-based refactoring.
- **Subsystem and system integration testing**
  A similar procedure is used for (sub-)system integration testing: Sections of the overall system are integrated with each other so that their interaction can be checked for correctness. Simulators and test frameworks may be necessary to do this. Usually, the components under test are not classes or modules, but rather multiple units that have already been packaged or that already form subsystems.

Robustness and data integrity

Integration testing is the test level of choice for verifying robustness and data integrity between units and components, as well as compliance with protocols and planned usage.

At this level, some degree of automation makes sense in many cases, as many systems that are only partially integrated don't yet have a user interface (or the user interface hasn't yet been integrated).

System Testing

In sequential development models, functional system testing takes place in its entirety at a certain point in the development process. Nevertheless, there are scenarios in which regression testing becomes necessary. These include:

- Subsequent change requests and enhancements
- Defect corrections
- Refactoring
- Redesign
- Maintenance

When test automation is mentioned in a testing context, it usually refers to automated system testing. To do this, many tools use the graphical user interface (GUI) and the database to automatically process test cases that were defined by testers and were previously performed manually. This is one of the reasons why automated system testing can be one of the most complex variants of test automation. Other reasons include:

- The focus is on the entire system under test—possibly including other underlying systems
- Test cases require business understanding
- GUIs are designed to interact with a human user, not with a program
- Mocks cannot be used to prepare sufficient test data, a task that needs to take place within the system itself
- Test driver creation for third-party system simulation is required if system integration testing is not planned

In many cases, automation is also an essential tool for non-functional system tests. Load and performance tests are simply not feasible without automation.

Acceptance Testing

In traditional, sequential development models, acceptance testing is performed after system testing at the end of a software development process. The software created is accepted by the customer based on the requirements documentation created at the beginning of the project.

Agile and iterative approaches

In some software development models, especially those that use agile and iterative approaches, concrete acceptance criteria in the form of test cases are defined in advance with the involvement of all stakeholders.

→

These then serve as the basis for determining whether an implemented functionality is considered complete.

Behavior-driven development

Behavior-driven development (BDD) [URL: BDD] is a technique that can be seen as an extension of test-driven development that includes automated verification of the fulfillment of acceptance criteria. BDD defines test cases in a way that makes them both automatable and comprehensible from a business point of view. This provides the developer with sufficient background information about the purpose of the expected code in the form of examples.

Domain-specific languages

For this purpose, domain-specific languages are defined for writing and recording test cases. Automating these test cases (or the creation of an automation environment that can process them automatically) is part of the development process and can be done in advance with the help of frameworks.

A typical BDD test case consists of three key elements: preconditions (i.e., a **given**), actions (i.e., **when** something happens), and verifications (i.e., **then** …).

For example:

- **Given**
  - A customer under 16 years of age is signed in
  - and the customer's shopping cart is empty
  - and an event has an age restriction for over-16s only
- **When** the customer puts this event in the cart
- **Then** Error message: *Event has an age restriction of 16* appears ...
- ... and no ticket should appear in the cart

In this example, the first section describes the required test data that must be provided before the test case is executed, the second section describes an action performed on the test object, and the third section specifies validations of the SUT's response and the corresponding expected results.

1.5.2 Test Automation Approaches for Different Types of Projects

Different project types require different

Different project types require a different approach when it comes to test automation. The automated testing of a standalone application may have limited technological scope and will focus on functional correctness. In addition, the focus is on regression testing of multiple planned releases. In contrast, the use of test automation in a data migration project is usually seen more as a consistency and comparison test that is not designed to be repeated for years to come.

→

A Conventional Software Development Project

As simple as this kind of project may seem, due to clear functional specifications and software testing documentation, implementing comprehensive test automation can nevertheless be quite tricky. However, this is not due to business or technical challenges, as extensive and well-structured business requirements and the corresponding test cases can usually be developed and made available in time for test execution. A bigger problem faced by the automation process is the timely availability of the application to be automated. When the availability of the test object is delayed, the developer often expects rapid test execution and feedback rather than the start of a potentially laborious automation process.

Conventional software development projects therefore initially prioritize manual testing and only consider test automation in cases where a project is planned for the long term with multiple shipping versions. However, automation is also occasionally employed when testing is strongly data-driven and testing large numbers of value combinations is required. In this case, the investment in automation may already pay off with a single test execution.

Maintenance and Product Enhancement Projects

Currently, software test automation is not very common in these types of projects, although structured test execution in general isn't either. The testing of smaller extensions and modifications is either performed by the developers themselves or delegated to the relevant business units. However, this kind of scenario is becoming increasingly difficult to sustain. Applications are becoming increasingly complex while test resources are increasingly limited and their costs transparent. If a business unit generates significant costs that are easily understood, it becomes increasingly difficult to justify the approach.

From an organizational point of view, this type of project therefore represents an ideal starting point for test automation. Once a basic automation strategy has been defined, it can be implemented in small steps on an ongoing basis. These steps allow for a continuous learning and adaptation cycle. Implementation can, for example, be prioritized according to the following benefits:

Automation steps in maintenance projects

- **Automation of test cases that affect current changes**. Newly developed areas and those affected by changes are much more prone to defects than areas of the application that have been in production for a long time. This remains true for several subsequent releases.

→

- **Automation of testing activities that are normally performed by the business unit.** In a first step, these tests are also implemented at a comparable level of quality and detail. This reduces effort for the business units, thus saving significant costs while maintaining the same (or a higher) level of quality.
- **Automation of test cases derived from the analysis of problems and failures during operation.** This source is particularly useful for selecting the test scenarios to be automated, especially when it comes to stabilizing the SUT.
- **Ongoing optimization and expansion** of the above scenarios and especially regression testing, which is normally performed manually by the business unit or the testing department.

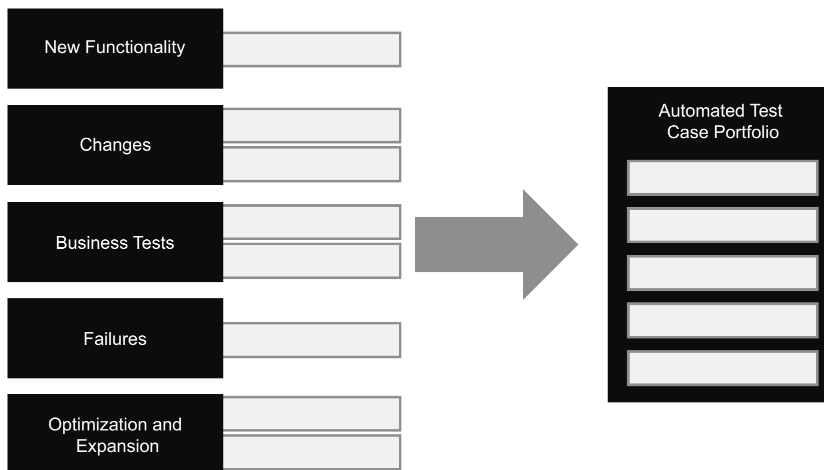


Fig. 1–7 *Sources of changes in the automated test case portfolio in maintenance projects*

SAP Projects

SAP implementations or enhancement projects are a special case. The ongoing release changes, upgrades, or enhancement packages pose a significant challenge to the company concerned. For every change, employees from the business units have to check the functionality of the system after the changes have been implemented. Customized settings, individual extensions and the system interfaces are particularly affected, and the testing effort required quickly reaches an almost unmanageable level. This is one of the reasons many companies don't install every upgrade or package and prefer to forgo system improvements rather than jeopardize overall system stability.

Automation in an SAP environment

It is therefore logical that the call for automated tests in this type of environment is becoming increasingly loud. The highly standardized application landscape and mostly uniform user interfaces and interface architectures make a good starting point for automation. Many manufacturers of commercial automation solutions are also SAP-certified and make direct use of transparent SAP technologies.

However, automation is still not standard in SAP environments, partly due to the challenges involved in test environments, the provision of test data, and the restoration of a consistent database for cross-system test execution. It is also due to the complex question of which of the thousands of possible tests should be automated. The right answers to these conceptual questions are key to successful automation in SAP projects.

Agile Projects

High-efficiency testing methods and pair testing

"Working software is the primary measure of progress", states one of the twelve principles of the *Agile Manifesto* [URL: AGILE]. The goal of each sprint is the availability of functional and potentially releasable (i.e., correct) software. However, especially in the case of very short development cycles, this is difficult to ensure if programming is carried out right up to the last minute. Test automation is essential if you want to be sure that all necessary testing activities can still be performed, especially in an agile environment. However, automation is complicated by the fact that there are often no stable test objects available, and the artifacts to be tested are subject to constant change. This applies not only to the application components that are implemented during a sprint, but also to those from previous iterations. "Welcome changing requirements, even late in development" is the corresponding principle in the *Agile Manifesto*.

This means that full regression tests must be performed regularly for quality assurance and especially for test automation. Furthermore, these tests also must be continuously adapted to new requirements and technical and/or functional changes. It is therefore no surprise that many agile projects work with completely different methods and approaches to those found in conventional projects.

Other important approaches that strongly shape the daily work of agile testers and developers are exploratory testing and pair testing (or pairing in general). However, since these methods are not directly supported by test automation, we suggest that interested readers refer to [Kaner et al. 02], [Baumgartner et al. 21] and [Linz 14].

Continuous integration and continuous delivery, as well as technical approaches such as test-driven development (TDD) and acceptance test-driven development (ATDD) are important in projects that use test automation.

→

Continuous integration and delivery

"Continuous delivery" doesn't actually describe a single technique or method, but rather a collection of principles that have the mutual goal of automating large parts of the integration and delivery process. In addition to comprehensive test automation (unit testing, system testing, and acceptance testing), this also includes continuous integration, automated provisioning of test systems, and automated delivery to different systems (development, QA, and production environments).

Martin Fowler briefly summarizes continuous delivery as follows [URL: Fowler].

"You're doing continuous delivery when:

- *Your software is deployable throughout its lifecycle*
- *Your team prioritizes keeping the software deployable over working on new features*
- *Anybody can get fast, automated feedback on the production readiness of their systems any time somebody makes a change to them*
- *You can perform push-button deployments of any version of the software to any environment on demand"*

Continuous integration is only one part of a continuous delivery process. Another central component is the automated execution of tests at various test levels, resulting in specific requirements for the test automation tools:

- Can test execution be integrated into a build system (such as *Maven* or *Ant*)? Is it possible for the automated test cases to be executed automatically with every build of the system and, depending on the result, to influence the continuing build process (for example, abort a build in case of failed test execution)?
- Is it possible to manage and display the test results of different test levels in a uniform manner? Since test automation for components also plays an important role (especially in agile projects), it is essential that these can be displayed and managed in the same way as automated integration or system tests.
- How can the automated test cases be executed within different environments? This question is particularly relevant if the software is to be delivered continuously to different target systems. This should then be possible without any additional effort—ideally by changing only a single configuration parameter.

In summary, the automation of functional regression testing is a must-have for agile projects. This applies both to the expansion of unit testing (especially using test-driven development) and to the automation of functional system tests and automated (or partially automated) acceptance tests.

→

To make this possible, you cannot afford to ignore the overall organizational conditions. In agile projects, this means that the automatability (i.e., the testability) of an application and the degree of automation for a user story are essential items in the team's "definition of done", and must be treated like any other acceptance criteria.

Automation is a must-have in agile projects

This is important because it creates an awareness of the steadily increasing testing effort that comes with each sprint. In practice, it is rare that all existing functionalities are re-tested in each sprint. Even if only some functionalities are regression tested, the balance between functionality to be tested and functionality to be developed within a sprint quickly shifts to the detriment of testing. The team must respond by driving automation and adjusting sprint planning accordingly. A good agile team will always find the right solution because the entire team is responsible for the functioning software that emerges at the end of the sprint.

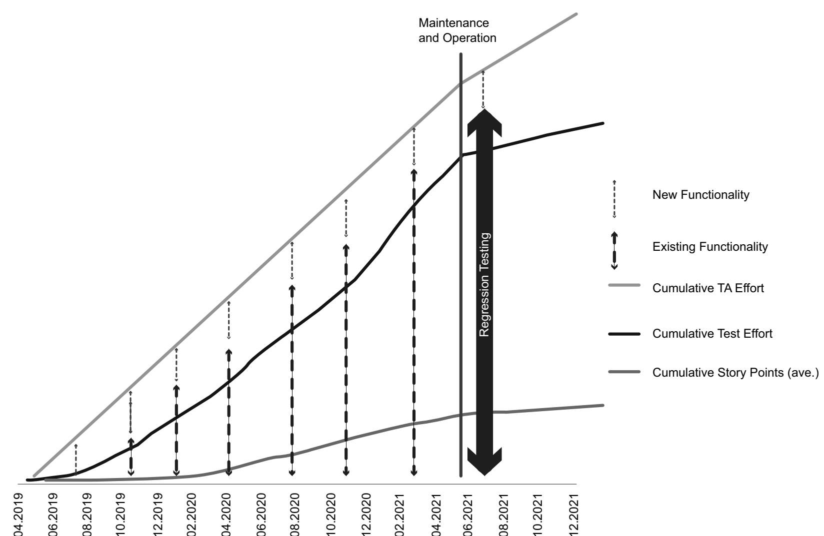


Fig. 1–8
Increasing effort and growth of the regression test portfolio in an agile project

DevOps

In addition to agile projects, DevOps is another software development concept that has enjoyed increasingly widespread use in recent years. The basic idea is very much based on agile principles and agile software development practices, but additional focus is placed on the integration of business units and operations as well as automation in all areas of the software lifecycle (development, testing, deployment, and operations). On an abstract level, the objectives of DevOps can be summarized as follows:

→

- Improvement of overall system quality and therefore added value for end-users
- Shorter delivery cycles and thus more effective feedback cycles
- Cost reduction and increased efficiency
- More flexibility and thus an increased ability to respond to changing conditions

The CALMS framework assesses a company's ability to adopt the DevOps approach, and its acronym represents the following organizational viewpoints.

Culture

An essential premise is that DevOps cannot be regarded simply as a process or an approach to software development. Instead, it revolves around the team culture and organizational culture, which are strictly oriented towards seamless, cross-functional collaboration.

In many cases, a shared team vision and mission can help to lay the foundations for this culture. There are various approaches to building such a vision, but a major prerequisite is trust between team members and between team members/DevOps and management. In turn, management also needs to have faith in the corresponding return on investment (ROI).

This open culture inevitably involves transparency in communication. Success and failure are part of the culture and both are part of the process. Failures should lead to the establishment of continuous improvement or to learning something new. The way a company deals with failures is often deeply ingrained in the organizational culture and can prevent the success of DevOps, for example through open or concealed apportioning of blame.

Automation

Generally speaking, organizations should strive to automate as many manual and recurring tasks as possible, but they need to consider stability, maintainability, and simplicity when doing so. The subject of automation covers a broad range of subtopics, many each of which could fill entire articles and books on their own. The following sections summarize some of the most important ones:

- **Automating the build process**
  As previously mentioned, continuous integration and delivery are core practices here, but the automation process also has to include an adequate branching concept and automated versioning (for example, semantic versioning) for build artifacts.
- **Automating the testing process**
  When automating, activities that are part of the testing process are an obvious place to start. It is important to emphasize that this applies to all test levels and test types, from requirements quality assurance, ensuring sufficient coverage through automated unit testing, static code analysis and automated validation of complete systems, all the way up to automated test activities in productive systems (for example, A/B testing). The detection of possible security risks and attack vectors can also be ensured using dynamic application security testing (DAST), static application security testing (SAST), dependency scanning, container scanning, and secret detection.
- **Automating infrastructure provisioning**
  The infrastructure as code (IaC) approach, using common tools such as *Ansible*, *Chef*, *Terraform*, *Puppet*, *Kubernetes* and others, is just as much part of this process as approaches such as GitOps, where it is especially clear how closely operations have to cooperate with other areas of the software lifecycle. In turn, this makes infrastructure a central development artifact, and makes the team responsible for ensuring its quality.
- **Automating the deployment and delivery process**
  In addition to important principles such as continuous delivery or continuous deployment, other questions also need to be addressed. These include automating the change log and version references and archiving artifacts to ensure traceability.
- **Automating the monitoring process**
  As previously mentioned, you need to consider how a product will eventually be used right at the start of the development process. Furthermore, in addition to monitoring the infrastructure, you also have to think about possible application-specific items. Analyzing log files or, more specifically, syntactically correct logging, is essential. Customer feedback and data collection from A/B or functional tests are also important monitoring tasks that have to be considered too.

Lean

Development teams use lean software development principles to eliminate "waste" by defining end-user value and understanding how to achieve it. For example, the value stream is optimized by minimizing concurrent work (using a WIP limit), making work and progress visible and traceable, reducing handoff complexity, and breaking down steps to ensure that the flow of remaining steps is smooth, uninterrupted and wait-free. It also includes introducing cross-functional teams and training employees to be versatile and adaptable.

Measurement

To better understand the capabilities and potential for optimization in the current system, it is necessary to have well-defined metrics. Data and information for collection and analysis can be planning data, product data, quality data, or more general team data. However, the basic premise is always that this data should not be used to monitor the team, but rather to continuously improve it. This is only possible if there is sufficient trust and an appropriate failure culture is established. Otherwise, you will have to assume that your metrics will be only partially valid, or not valid at all.

In terms of continuous improvement, the following activities are therefore helpful:

- Collect and analyze product and system-specific data
- Define metrics and thresholds
- Monitor and track metrics, and automate notifications
- Detect and document failures
- Define quality gates and ensure that they are complied with
- Create a culture of continuous learning and improvement
- Improve efficiency and reduce cycle times

Sharing

Typically, this involves establishing a blame-free culture, which may sound simple, but requires plenty of experience and understanding as well as good role models at management level.

An open communication culture should also promote the principle of asking and sharing. Good technical/organizational solutions and experiences should be shared within and between teams. This helps to transfer the resulting improved efficiency to other parts of the organization.

Migration Projects

For migration projects, the million-dollar question is: does the system still function as it did before? Test automation can help to answer this question in several ways.

Test Data Generation

To test data migration at an early stage, you need to prepare two test data sets:

- **Machine-generated synthetic data**
  This data set is based on the migration rules and thus tests the implemented data import procedures in a structured way. It is derived from test case specification methods and can therefore be generated using the available tools.
- **Production data**
  Tests with production data are essential, as this is the data that will ultimately be migrated. There are always real-world data configurations that are not specified in the requirements or design documents, and that cause problems during migration. Here, automation can be used to export the production data from the old system and, if necessary, anonymize it. The exported data can also be used post-test for the actual migration. Anonymization may be required for legal reasons (for example, to prevent the test team from seeing personal data) or due to general data privacy requirements. However, it is important to ensure that anonymized data retains the original data structure and doesn't corrupt specific attributes such as spelling.

The task of test data management is a very complex one and especially the handling of sensitive data and large amounts of data for test and test automation purposes requires a lot of knowledge and experience. These circumstances motivated the German Testing Board to develop a training course for the *Certified Test Data Specialist (GTB)*, whose curriculum provides a good overview of this topic (only in German) [GTB: TDS].

Data Comparison

Newly migrated data cannot be manually checked against the original data set, especially when large amounts of data are migrated. Data comparison therefore has to be automated. This can, for example, be performed using specialized comparator tools that can also apply certain specific rules to the data sets being compared.

→

Process Comparison

The feasibility of using test automation to check functionality before and after a migration depends very much on the type of migration. For example, if an application has been migrated to a new platform, or if it has been connected to a different database, running a comprehensive, automated test suite that has already been developed for the previous system can very quickly deliver the desired results in the new environment. This task is more complex if a system has been automatically transformed—for example, from Cobol to Java. In most cases the automated test cases must be adapted, and the effort required depends on the extent to which the business test cases are decoupled from the technical implementation.

If the migration is to a completely new solution, an automated comparison of processes with those in the original system may not be practical from a cost-benefit point of view.

Migrating a Test Automation Solution

Just like the system itself, an existing test automation solution can also be migrated. If the automation in question is keyword-based and the processes remain the same, it is sufficient to change the scripts behind the keywords. New keywords may have to be created or old keywords declared obsolete, but the description language remains the same. This means that testers do not have to learn a new language and can still use most of the test cases that have already been automated.

Real-World Examples:
Gradual conversion of keywords

In an agile legacy system replacement project, keyword-based test cases were implemented. Due to the complex workflows, some actions had to be performed on the legacy system to implement automated end-to-end test cases. During the project, the legacy system was replaced step by step, and the keywords were changed one by one to address the newly developed product instead of the legacy system. Due to the similar structures and workflows of the two systems, it was possible to retain a large portion of the existing test cases, even though the technologies and activities performed were ultimately very different.