In software testing, test automation is the use of software separate from the software being tested to control the execution of tests and the comparison of actual outcomes with predicted outcomes.[1] Test automation can automate some repetitive but necessary tasks in a formalized testing process already in place, or perform additional testing that would be difficult to do manually. Test automation is critical for continuous delivery and continuous testing.[2]

There are many approaches to test automation, however below are the general approaches used widely:

Graphical user interface testing. A testing framework that generates user interface events such as keystrokes and mouse clicks, and observes the changes that result in the user interface, to validate that the observable behavior of the program is correct.

API driven testing. A testing framework that uses a programming interface to the application to validate the behaviour under test. Typically API driven testing bypasses application user interface altogether. It can also be testing public (usually) interfaces to classes, modules or libraries are tested with a variety of input arguments to validate that the results that are returned are correct.

One way to generate test cases automatically is model-based testing through use of a model of the system for test case generation, but research continues into a variety of alternative methodologies for doing so.[citation needed] In some cases, the model-based approach enables non-technical users to create automated business test cases in plain English so that no programming of any kind is needed in order to configure them for multiple operating systems, browsers, and smart devices.[3]

What to automate, when to automate, or even whether one really needs automation are crucial decisions which the testing (or development) team must make.[4] A multi-vocal literature review of 52 practitioner and 26 academic sources found that five main factors to consider in test automation decision are: 1) System Under Test (SUT), 2) the types and numbers of tests, 3) test-tool, 4) human and organizational topics, and 5) cross-cutting factors. The most frequent individual factors identified in the study were: need for regression testing, economic factors, and maturity of SUT.[5]

A growing trend in software development is the use of unit testing frameworks such as the xUnit frameworks (for example, JUnit and NUnit) that allow the execution of unit tests to determine whether various sections of the code are acting as expected under various circumstances. Test cases describe tests that need to be run on the program to verify that the program runs as expected.

Test automation, mostly using unit testing, is a key feature of extreme programming and agile software development, where it is known as test-driven development (TDD) or test-first development. Unit tests can be written to define the functionality before the code is written. However, these unit tests evolve and are extended as coding progresses, issues are discovered and the code is subjected to refactoring.[6] Only when all the tests for all the demanded features pass is the code considered complete. Proponents argue that it produces software that is both more reliable and less costly than code that is tested by manual exploration.[citation needed] It is considered more reliable because the code coverage is better, and because it is run constantly during development rather than once at the end of a waterfall development cycle. The developer discovers defects immediately upon making a change, when it is least expensive to fix. Finally, code refactoring is safer when unit testing is used; transforming the code into a simpler form with less code duplication, but equivalent behavior, is much less likely to introduce new defects when the refactored code is covered by unit tests.

Some software testing tasks (such as extensive low-level interface regression testing) can be laborious and time-consuming to do manually. In addition, a manual approach might not always be effective in finding certain classes of defects. Test automation offers a possibility to perform these types of testing effectively.

Once automated tests have been developed, they can be run quickly and repeatedly many times. This can be a cost-effective method for regression testing of software products that have a long maintenance life. Even minor patches over the lifetime of the application can cause existing features to break which were working at an earlier point in time.

While the reusability of automated tests is valued by software development companies, this property can also be viewed as a disadvantage. It leads to the so-called "Pesticide Paradox", where repeatedly executed scripts stop detecting errors that go beyond their frameworks. In such cases, manual testing may be a better investment. This ambiguity once again leads to the conclusion that the decision on test automation should be made individually, keeping in mind project requirements and peculiarities.

Test automation tools can be expensive and are usually employed in combination with manual testing. Test automation can be made cost-effective in the long term, especially when used repeatedly in regression testing. A good candidate for test automation is a test case for common flow of an application, as it is required to be executed (regression testing) every time an enhancement is made in the application. Test automation reduces the effort associated with manual testing. Manual effort is needed to develop and maintain automated checks, as well as reviewing test results.

In automated testing, the test engineer or software quality assurance person must have software coding ability since the test cases are written in the form of source code which when run produce output according to the assertions that are a part of it. Some test automation tools allow for test authoring to be done by keywords instead of coding, which do not require programming.
API testing

API testing is also being widely used by software testers as it enables them to verify requirements independent of their GUI implementation, commonly to test them earlier in development, and to make sure the test itself adheres to clean code principles, especially the single responsibility principle. It involves directly testing APIs as part of integration testing, to determine if they meet expectations for functionality, reliability, performance, and security.[7] Since APIs lack a GUI, API testing is performed at the message layer.[8] API testing is considered critical when an API serves as the primary interface to application logic.[9]
Continuous testing

Continuous testing is the process of executing automated tests as part of the software delivery pipeline to obtain immediate feedback on the business risks associated with a software release candidate.[10][11] For Continuous Testing, the scope of testing extends from validating bottom-up requirements or user stories to assessing the system requirements associated with overarching business goals.[12]
Graphical User Interface (GUI) testing
Main article: Graphical user interface testing

Many test automation tools provide record and playback features that allow users to interactively record user actions and replay them back any number of times, comparing actual results to those expected. The advantage of this approach is that it requires little or no software development. This approach can be applied to any application that has a graphical user interface. However, reliance on these features poses major reliability and maintainability problems. Relabelling a button or moving it to another part of the window may require the test to be re-recorded. Record and playback also often adds irrelevant activities or incorrectly records some activities.[citation needed]

A variation on this type of tool is for testing of web sites. Here, the "interface" is the web page. However, such a framework utilizes entirely different techniques because it is rendering HTML and listening to DOM Events instead of operating system events. Headless browsers or solutions based on Selenium Web Driver are normally used for this purpose.[13][14][15]

Another variation of this type of test automation tool is for testing mobile applications. This is very useful given the number of different sizes, resolutions, and operating systems used on mobile phones. For this variation, a framework is used in order to instantiate actions on the mobile device and to gather results of the actions.

Another variation is script-less test automation that does not use record and playback, but instead builds a model[clarification needed] of the application and then enables the tester to create test cases by simply inserting test parameters and conditions, which requires no scripting skills.
Testing at different levels

A strategy to decide the amount of tests to automate is the test automation pyramid. This strategy suggests to write three types of tests with different granularity. The higher the level, less is the amount of tests to write.[16]
Unit, Service, and UI levels
The test automation pyramid proposed by Mike Cohn[16]

As a solid foundation, unit testing provides robustness to the software products. Testing individual parts of the code makes it easy to write and run the tests. Developers write unit tests as a part of each story and integrate them with CI.[17]
The service layer refers to testing the services of an application separately from its user interface, these services are anything that the application does in response to some input or set of inputs.
At the top level we have UI testing which has fewer tests due to the different attributes that make it more complex to run, for example the fragility of the tests, where a small change in the user interface can break a lot of tests and adds maintenance effort.[16][18]

Unit, Integration, and End-to-End levels
A triangular diagram depicting Google's "testing pyramid". Progresses from the smallest section "E2E" at the top, to "Integration" in the middle, to the largest section "Unit" at the bottom.
Google's testing pyramid[19]

One conception of the testing pyramid contains unit, integration, and end-to-end unit tests. According to Google's testing blog, unit tests should make up the majority of your testing strategy, with fewer integration tests and only a small amount of end-to-end tests.[19]

Unit tests: These are tests that test individual components or units of code in isolation. They are fast, reliable, and isolate failures to small units of code.
Integration tests: These tests check how different units of code work together. Although individual units may function properly on their own, integration tests ensure that they operate together coherently.
End-to-end tests: These test the system as a whole, simulating real-world usage scenarios. They are the slowest and most complex tests.

Framework approach in automation

A test automation framework is an integrated system that sets the rules of automation of a specific product. This system integrates the function libraries, test data sources, object details and various reusable modules. These components act as small building blocks which need to be assembled to represent a business process. The framework provides the basis of test automation and simplifies the automation effort.

The main advantage of a framework of assumptions, concepts and tools that provide support for automated software testing is the low cost for maintenance. If there is change to any test case then only the test case file needs to be updated and the driver Script and startup script will remain the same. Ideally, there is no need to update the scripts in case of changes to the application.

Choosing the right framework/scripting technique helps in maintaining lower costs. The costs associated with test scripting are due to development and maintenance efforts. The approach of scripting used during test automation has effect on costs.

Various framework/scripting techniques are generally used:

Linear (procedural code, possibly generated by tools like those that use record and playback)
Structured (uses control structures - typically ‘if-else’, ‘switch’, ‘for’, ‘while’ conditions/ statements)
Data-driven (data is persisted outside of tests in a database, spreadsheet, or other mechanism)
Keyword-driven
Hybrid (two or more of the patterns above are used)
Agile automation framework

The Testing framework is responsible for:[20]

- defining the format in which to express expectations
- creating a mechanism to hook into or drive the application under test
- executing the tests
- reporting results

Test automation interface

Test automation interfaces are platforms that provide a single workspace for incorporating multiple testing tools and frameworks for System/Integration testing of application under test. The goal of Test Automation Interface is to simplify the process of mapping tests to business criteria without coding coming in the way of the process. Test automation interface are expected to improve the efficiency and flexibility of maintaining test scripts.[21]
Test Automation Interface Model

Test Automation Interface consists of the following core modules:

- Interface Engine
- Interface Environment
- Object Repository

Interface engine

Interface engines are built on top of Interface Environment. Interface engine consists of a parser and a test runner. The parser is present to parse the object files coming from the object repository into the test specific scripting language. The test runner executes the test scripts using a test harness.[21]
Object repository

Object repositories are a collection of UI/Application object data recorded by the testing tool while exploring the application under test.[21]
Defining boundaries between automation framework and a testing tool

Tools are specifically designed to target some particular test environment, such as Windows and web automation tools, etc. Tools serve as a driving agent for an automation process. However, an automation framework is not a tool to perform a specific task, but rather infrastructure that provides the solution where different tools can do their job in a unified manner. This provides a common platform for the automation engineer.

There are various types of frameworks. They are categorized on the basis of the automation component they leverage. These are:

- Data-driven testing
- Modularity-driven testing
- Keyword-driven testing
- Hybrid testing
- Model-based testing
- Code-driven testing
- Behavior driven development

What to test

Testing tools can help automate tasks such as product installation, test data creation, GUI interaction, problem detection (consider parsing or polling agents equipped with test oracles), defect logging, etc., without necessarily automating tests in an end-to-end fashion.

One must keep satisfying popular requirements when thinking of test automation:

Platform and OS independence
Data driven capability (Input Data, Output Data, Metadata)
Customization Reporting (DB Data Base Access, Crystal Reports)
Easy debugging and logging
Version control friendly – minimal binary files
Extensible & Customization (Open APIs to be able to integrate with other tools)
Common Driver (For example, in the Java development ecosystem, that means Ant or Maven and the popular IDEs). This enables tests to integrate with the developers' workflows.
Support unattended test runs for integration with build processes and batch runs. Continuous integration servers require this.
Email Notifications like bounce messages
Support distributed execution environment (distributed test bed)
Distributed application support (distributed SUT)