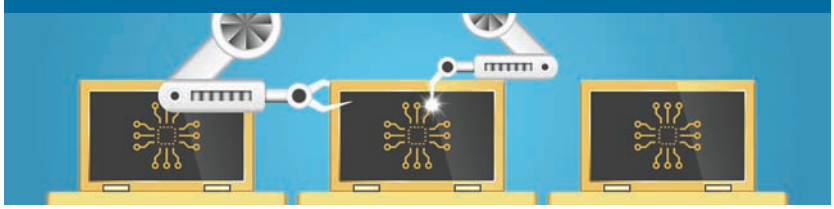
Test Automation

Not Just for Test Execution

Vahid Garousi, Hacettepe University

Frank Elberzhager, Fraunhofer Institute for Experimental Software Engineering

// To work more efficiently and effectively, test engineers must be aware of various automated-testing strategies and tools that assist test activities other than just test execution. However, automation doesn't come for free, so it must be carefully implemented. //



SOFTWARE TESTING IS costly and effort-intensive. According to Pierre Audoin Consultants, worldwide testing costs (including testing software, hardware, and services) were 79 billion euros in 2010 and were expected to exceed 100 billion euros in 2014.[1] Recent hot topics, such as agile methods and continuous integration, strongly motivate software engineers to think about how to further automate testing during development, deployment, and delivery, beyond mere test execution.

However, software engineers will be able to fully benefit from such automation only if they're aware of the available strategies and tools. This article aims to increase this awareness, describe test automation's benefits, and demonstrate that industrial testers should broaden their definition of test automation beyond only test execution. (For a discussion of the state of test automation, see the related sidebar.)

Automation across the Software-Testing Process

Typically, a test process comprises several steps from planning to test specification, execution, and reporting.[2] To better understand how this process can employ automation, we derived six testing activities with a large potential for automation:

1. Test-case design.
2. Test scripting.
3. Test execution.
4. Test evaluation.
5. Test results reporting.
6. Test management and other test-engineering activities.

Figure 1 shows an overview of these activities. Each of the first five activities can be executed manually (by a human), automated (using a software tool), or a combination of both. (For a brief discussion of manual versus automated testing, see the related sidebar.)

Next, we discuss each of the six activities. Many tools exist that serve different purposes and support different testing activities (for examples, visit www.opensourcetesting.org or www.pairwise.org). Here, we present only a few examples. We aim not to provide a full catalog of tools for each activity but to stress automation's importance in test activities.

Test-Case Design

This activity outputs a suite of test cases (input and expected output values) or test requirements (for example, control flow paths to cover). The next testing phase, test scripting, will use this output. Test-case design can be based either on human knowledge (for example, exploratory testing) or on criteria (for example, line or requirements coverage).[2]

In manual design, the tester looks at the requirements or code (depending on whether the testing is black- or white-box) and derives the test cases without using any tool. Test-case design based on human knowledge is always manual.

For partial automation, the tester can use any of the many available code coverage tools but manually derives test cases to increase coverage.

0740-7459/16/$33.00 © 2016 IEEE

THE STATE OF TEST AUTOMATION

Test automation has a history of over two and half decades.[1] It originally dealt primarily with test execution, and the software industry's use of automated test execution is mature. There are many advanced and powerful tools for automated text execution—for example, Selenium and Testdroid.

Test automation has expanded to other testing activities. Large companies such as Microsoft and Google have benefited from such automation.[2,3] In discussions with several test engineers in Turkey, we observed that many practitioners practice end-to-end automation, from test-case design to execution and defect reporting.

However, the state of the practice regarding automation of testing activities other than test execution seems immature, although various tools and techniques in academia target these needs. For example, prototype tools such as EvoSuite[4] and AUSTIN (*Au*gmented *S*earch-Based *Test*i*n*g)[5] perform search-based test data generation. Nevertheless, we see only limited use of those tools in industry—for example, in the automotive sector.

Also, the adoption of test automation varies among companies. For example, one company might automate only some main test cases of a web application, whereas another company automates most of its testing. According to three surveys[6–8] and our experiences with more than 40 testing projects in close partnership with companies in Canada, Turkey, and Germany, many test engineers think of automation only in terms of tools for executing test cases—for example, GUI record-and-playback tools. In particular, small and medium-sized enterprises are often unaware of the great potential of automation throughout the testing life cycle or don't seriously consider greater use of automation.

Even Wikipedia seems to have a slightly limited definition of test automation: "In software testing, test automation is the use of special software (separate from the software being tested) to control the execution of tests and the comparison of actual outcomes with predicted outcomes."[9]

To further investigate the state of test automation, we examined books on it. To keep the workload manageable, we selected books published between 2010 and 2014 (the list is at http://goo.gl/8aTr4x). We checked whether each book discussed only test execution or also described other testing activities. Of the 78 books, only a handful discussed automation beyond test execution.

To increase the level of automation across all testing activities and encourage wider use of existing test tools for more than test automation, we need more research by and collaboration between researchers and practitioners.

References

1. K.C. Archie et al., *Test Automation System*, US patent 5,021,997, Patent and Trademark Office, 1991.
2. A. Page and K. Johnston, *How We Test Software at Microsoft*, Microsoft Press, 2008.
3. J.A. Whittaker, J. Arbon, and J. Carollo, *How Google Tests Software*, Addison-Wesley Professional, 2012.
4. G. Fraser and A. Arcuri, "EvoSuite: Automatic Test Suite Generation for Object-Oriented Software," *Proc. 19th ACM SIGSOFT Symp. and 13th European Conf. Foundations of Software Eng.* (ESEC/FSE 11), 2011, pp. 416–419.
5. K. Lakhotia, M. Harman, and H. Gross, "AUSTIN: A Tool for Search Based Software Testing for the C Language and Its Evaluation on Deployed Automotive Systems," *Proc. 2nd Int'l Symp. Search Based Software Eng.* (SSBSE 10), 2010, pp. 101–110.
6. V. Garousi and J. Zhi, "A Survey of Software Testing Practices in Canada," *J. Systems and Software*, vol. 86, no. 5, 2013, pp. 1354–1376.
7. V. Garousi and T. Varma, "A Replicated Survey of Software Testing Practices in the Canadian Province of Alberta: What Has Changed from 2004 to 2009?," *J. Systems and Software*, vol. 83, no. 11, 2010, pp. 2251–2262.
8. V. Garousi et al., "A Survey of Software Engineering Practices in Turkey," *J. Systems and Software*, Oct. 2015, pp. 148–177.
9. "Test Automation," *Wikipedia*, 2016; https://en.wikipedia.org/wiki/Test_automation.

For conducting fully automated criteria-based test-case design, a large body of knowledge is available. An important part of this process is search-based test data generation, including inputs and expected outputs. Also, for automating combinatorial criteria-based approaches such as pairwise testing,[2] open-source and commercial tools exist that generate test inputs; one such tool is Hexawise.

Another tool is Crawljax, an open-source Java tool for automatically crawling and testing Ajax-based web

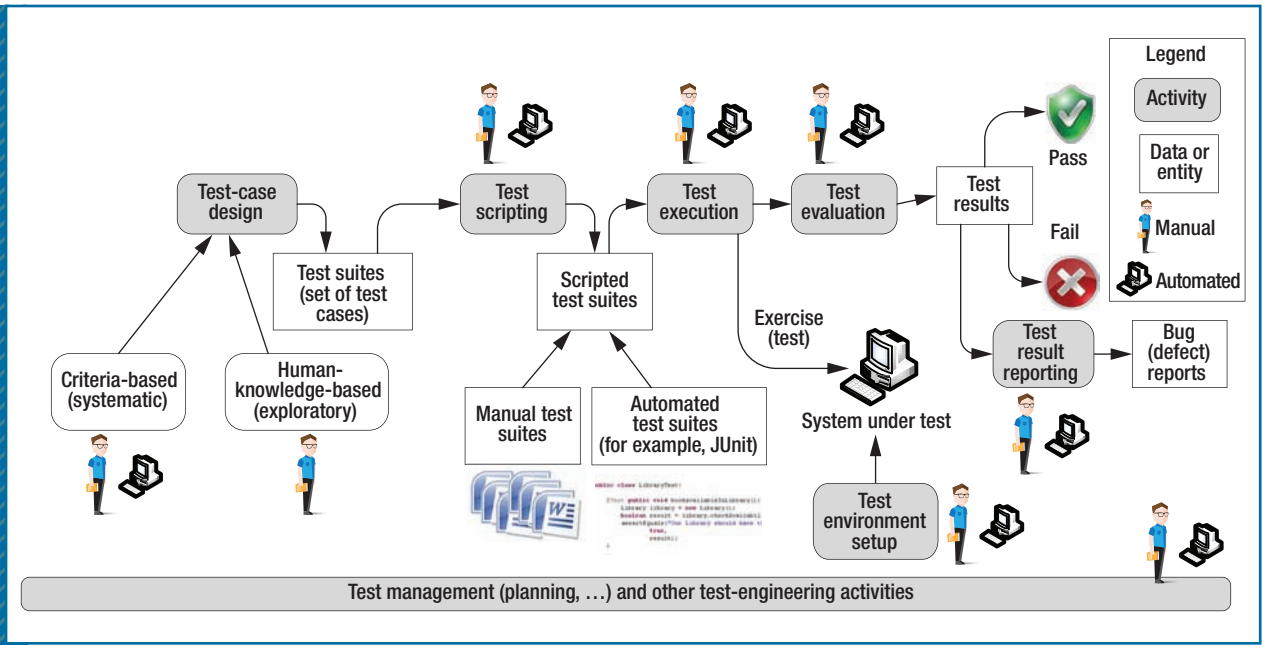


FIGURE 1. Automation across the software-testing process. Each testing activity offers tremendous opportunity for automation.

MANUAL VERSUS AUTOMATED TESTING

In manual testing, the tester assumes the role of a user executing the system under test (SUT) to verify its behavior and find any observable defects. In automated testing, developers develop test code scripts (for example, using the JUnit framework) that execute without human intervention to test the SUT's behavior. If planned and implemented properly, automated testing can yield various benefits over manual testing, such as repeatability and reduced test costs (and thus effort). However, if not implemented properly, automated testing will increase costs and effort and could even be less effective than manual testing.[1]

Reference

1. Y. Amannejad et al., "A Search-Based Approach for Cost-Effective Software Test Automation Decision Support and an Industrial Case Study," *Proc. Int'l Workshop Regression Testing*, 2014, pp. 302–311.

applications. Crawljax employs an event-driven dynamic crawling engine. It automatically creates a state-flow graph of the dynamic states in the Document Object Model and the event-based transitions between them. This graph is then used for automating many types of web analysis and testing techniques—for example, test-case generation and test scripting.

Perhaps one of the best success stories of automating test-case design is Microsoft's SpecExplorer tool. It uses model-based techniques to automatically generate test scripts and has been empirically evaluated.[3]

Test Scripting

Test scripting outputs either manual test scripts (in a variety of formats) or automated test suites for use in test execution. Testers have been manually performing test scripting for many years.

For partial automation, tools such as IBM Rational Manual Tester document and store test cases for later manual execution. Furthermore, many tools let developers test through GUIs; such tools are mostly called record-and-playback tools—for example, HP QuickTest Professional. With them, testers record a test scenario (test case) by interacting with the GUI of the system under test (SUT), while the tool

automatically records the test log as a test script in the background. After this recording, the test code can be executed as many times as needed without tester intervention.

For fully automated test scripting, good progress has been made in the last decade, and many tools for automated generation of automated test code are available.[4–6]

For example, a team from IBM Research India reported that conventional test automation techniques, such as record-and-playback and keyword-driven automation, can be time-consuming, can require specialized skills, and can produce fragile scripts.[7] To address these limitations, the team developed TACT (*T*ools for *A*utomated and *C*ost-Effective *T*esting), which automates test automation.[8] TACT (previously called ATA—automating test automation) uses a novel combination of natural-language processing, backtracking exploration, runtime interpretation, and learning to help testers efficiently create automated test scripts from manual test cases. Several case studies have successfully used the tool—for example, in various IBM projects.[7]

EvoSuite automatically generates JUnit test suites to satisfy a coverage criterion for Java programs. Similarly, Microsoft Pex automatically generates NUnit test suites.

Later in this article, we describe a tool that Shahnewaz Jolly and her colleagues created for automatically generating scripts, and present an example test script. Vahid Garousi and Michael Felderer have reviewed other tools and test patterns for automating test scripting.[6]

Test Execution

Test execution runs the test cases on the SUT and records the results or observes the SUT's output or behavior.[2] Decisions made during the previous activities affect test execution. For example, if a tester develops all the tests as automated test suites, test execution will obviously be fully automated. In contrast, if the test team develops all its tests as manual test scripts, test execution must be manual. Test execution is partially automated if some scripts are automated and others are manual.

Many automated test execution tools exist. For example, see the *IEEE Software* article "Test Automation"[9] or the online lists at www.opensourcetesting.org and en.wikipedia.org/wiki/List_of_unit_testing_frameworks.

Test Evaluation

There are usually three approaches for evaluating the test outcome (pass or fail):

- A human tester makes the judgment.
- The developers incorporate (hard-code) test evaluations as verification points (assertions) in the test code.
- The developers build "intelligent" (learning) test oracles, using machine learning and AI.[10]

The second and third approaches automate test evaluation, but with different levels of intelligence. Furthermore, developers sometimes write assertions in production code, which are called code contracts, following the design-by-contract methodology.

Test Results Reporting

This is usually the last phase; it reports test verdicts and defects to the developers for fixing—for example, through defect-tracking systems. This activity is usually manual, but techniques and frameworks exist for automated bug reporting. For example, the NBug .NET library automatically creates and sends bug reports, crash reports with a minidump, and error or exception reports with a stack trace.

Test Management and Other Test-Engineering Activities

These activities include test set minimization (which includes test redundancy detection), regression test selection, and test repair (conducting comaintenance on broken automated tests when the SUT has changed).

The CodeCover test coverage tool produces an autogenerated redundancy graph of a JUnit test case. This graph helps testers, in a semiautomated manner, detect redundant tests and minimize test sets.

The xRegress commercial tool considers code coverage in selecting the smallest set of regression tests.

For test repair, there are tools in academia but not many industrial-strength tools. One such tool is ReAssert, which automatically suggests repairs for broken unit tests.

Later in this article, we describe a genetic-algorithm-based tool that helps users determine when to automate.

Two Case Studies

The following case studies illustrate the successful automation of test phases other than execution. They show that such automation can offer considerable savings.

Test Scripting

A Canadian firm that develops control software was automating test scripting using a custom-developed tool for testing supervisory control and data acquisition (SCADA) software. The SUT, called Rocket

```
[TestClass]
 public class PowerFBTest
 {
 TestEngine TE = new TestEngine();
 static String FunctionBlock = "PowerFunctionBlock";
 static String FunctionBlockName = "Power1";
 […]
 [TestMethod]
  public void Testdc045ec99db541068f4aa5fc3cb1ad0b()
  {
   TE.setInputParameter(FunctionBlockName, "Base", "Int (8 bit)", "0");
   TE.setInputParameter(FunctionBlockName, "Exponent", "Int (8 bit)", "0");

   RocketParameter resultParam = TE.setOutputParameter(FunctionBlockName, "Result",
                                                       "Float (32 bit)");
   RocketParameter errorParam = TE.setOutputParameter(FunctionBlockName, "Error",
                                                      "Text");

   /* execution of the functionblock */
   TE.execute(FunctionBlock, FunctionBlockName);

   //test oracle - parameter: expected output, actual output */
   Assert.AreEqual("1", TE.getOutputByName(resultParam.PointName));
   Assert.AreEqual("", TE.getOutputByName(errorParam.PointName));
  }
```

FIGURE 2. An automatically generated NUnit test script. Automation saved the testers from manually writing approximately 20 KLOC of NUnit test code in this industrial case study.

Monitoring and Control, was developed with Microsoft Visual Studio C#. It had an Automation Engine tool, which was an IDE for developing advanced control systems. This tool supported 89 function blocks in 12 categories such as math, logic, and control. For example, the Add function block adds two or more input values and calculates the sum. The input can be any integer or floating-point value, or text.

If we consider three inputs to the Add Function unit in the SUT and apply multidimensional category partitioning (without pairwise testing), we get $7^3 = 343$ test cases. Assume that automating each test case in NUnit, for example, requires at least 4 test lines of code (TLOC)—one line for each xUnit test phase: setup, exercise, verify, and teardown. This results in $343 \times 4 = 1{,}372$ TLOC, and that's for only one of the 89 units under test. This highlights the magnitude of the effort for manually writing the test scripts.

The project team found that none of the existing automated test-code-generation tools was applicable to the SUT. So, the team developed and used AutoBBUT (Automated Black-Box Unit Testing), which automatically generates NUnit test suites.[11] Figure 2 shows a test script this tool generated. This approach saved the testers from manually writing approximately 20 KLOC of NUnit test code.

The team conducted cost–benefit measurements of the tool's use, employing precise time logging. Here are the results:

- *Benefits* (time savings) = 87 h (initial development of test code without the tool) + 87 h × 2 (test code maintenance without the tool) = 261 h
- *Cost* = 120 h (development) + 3 h (test code inspection and completion) = 123 h
- *Benefits* – *Cost* = 261 h – 123 h = 138 h (net time savings)

Since this project, the software development firm has been using the tool in its daily test activities.

Test Planning and Test-Case Design

This case dealt with the questions of what and when to automate in testing.[12,13] A Canadian software company was developing and testing embedded software for the oil and gas industry. In an action-research project, a team of practitioners and researchers developed a search-based approach to determine which system parts and which test cases and test activities to perform manually or automatically to ensure the highest return on investment (ROI) for test automation. This is one of the few systematic, measurement-driven decision-support techniques we know of.

To enable systematic decision making, the project team defined a Test Automation Decision Matrix (TADM), which indicates which test activities should be automated and

which should be manual. Table 1 shows an example TADM. To find the optimal TADM for a given test project, the team developed and used a genetic algorithm. By automating the activities the decision-support tool recommended, the company achieved an ROI of approximately 341 percent, without decreased test effectiveness (fault detection power).[12]

Furthermore, the team conducted automated test-case design using Hexawise.

TABLE 1

Table 1. A Test Automation Decision Matrix.[12]

Use case	Test-case design	Test scripting	Test execution	Test evaluation
1	Automated	Manual	Manual	Manual
2	Automated	Automated	Automated	Manual
3	Automated	Manual	Manual	Automated
…	…	…	…	…
N	Manual	Automated	Automated	Automated

Regarding trends such as agile development and continuous integration and delivery, a high degree of automation is needed to respond quickly to customer requests while providing high quality at reasonable costs. So, we encourage development companies, especially small and medium-sized enterprises, to evaluate their test automation potential and take steps toward increased automation.

Of course, every environment must be clearly analyzed regarding the potential test effectiveness and efficiency improvement that test automation provides. Any improvements must also be implemented with the needed rigor. Finally, such experiences should be shared—not only to learn from failed or challenged cases but also, and even more so, to convince other companies with success cases.

Automation doesn't come for free and thus must be carefully implemented to ensure success. For almost all the test activities we described, a large set of tools already exists for different environments. However, the tool market is fragmented. For example, many tools exist for different programming languages, environments, or IDEs and fall under different licensing types (open source, free, or commercial). Planning and implementing a proper test automation strategy in such a fragmented context is necessary and not trivial. This topic has been getting more attention in the testing industry (for example, see sqta.wordpress.com/test-automation-strategy) and will be one direction of our future research.

Regarding test automation strategy, the need exists for further research and empirical studies dealing with questions such as these:

- How much automation is enough? When and what (test cases) should you automate in the context of a given project?
- Which test phases should you automate, and which should be manual?
- Which test tool should you use, when should you use it, and under what conditions?
- Which test cases should you execute manually?
- What prerequisites should be met before you automate testing?

References

1. N. Mayes, "Software Testing Spends to Hit EUR100BN by 2014," Pierre Audoin Consultants, 2014; www.pac-online.com/software-testing-spends-hit-eur100bn-2014-press-release.
2. P. Ammann and J. Offutt, *Introduction to Software Testing*, Cambridge Univ. Press, 2008.
3. W. Grieskamp et al., "Model-Based Quality Assurance of Protocol Documentation: Tools and Methodology," *Software Testing, Verification and Reliability*, vol. 21, no. 1, 2011, pp. 55–71.
4. S.R. Shahamiri, W.M.N.W. Kadir, and S.Z. Mohd-Hashim, "A Comparative Study on Automated Software Test Oracle Methods," *Proc. Int'l Conf. Software Eng. Advances* (ICSEA 09), 2009, pp. 140–145.
5. D.J. Mosley and B.A. Posey, *Just Enough Software Test Automation*, Prentice Hall Professional, 2002.
6. V. Garousi and M. Felderer, "Developing, Verifying, and Maintaining High-Quality Automated Test Scripts," *IEEE Software*, vol. 33, no. 3, 2016, pp. 68–75.
7. S. Thummalapenta et al., "Automating Test Automation," *Proc. 34th Int'l Conf. Software Eng.* (ICSE 12), 2012, pp. 881–891.
8. "TACT: Tools for Automated and Cost-Effective Testing," IBM Research India; http://researcher.watson.ibm.com/researcher/view_group_subpage.php?id=3614.
9. M. Polo et al., "Test Automation," *IEEE Software*, vol. 30, no. 1, 2013, pp. 84–89.
10. K. Frounchi et al., "Automating Image Segmentation Verification

ABOUT THE AUTHORS

VAHID GAROUSI is an associate professor of software engineering at Hacettepe University and a senior consultant at Maral Software Consulting Corporation. His research interests include software testing, empirical software engineering, and improving industry–academia collaboration in software engineering. Garousi received a PhD in software engineering from Carleton University. He was an IEEE Computer Society Distinguished Visitor from 2012 to 2015. Contact him at vahid.garousi@hacettepe.edu.tr.

FRANK ELBERZHAGER is a senior engineer at the Fraunhofer Institute for Experimental Software Engineering. His research interests include software quality assurance, inspection and testing, software engineering processes, and software architecture. He also transfers research results into practice. Elberzhager received a PhD in computer science from the University of Kaiserslautern. Contact him at frank.elberzhager@iese.fraunhofer.de.

and Validation by Learning Test Oracles," *Information and Software Technology*, vol. 53, no. 12, 2011, pp. 1337–1348.

11. S.A. Jolly, V. Garousi, and M.M. Eskandar, "Automated Unit Testing of a SCADA Control Software: An Industrial Case Study Based on Action Research," *Proc. IEEE 5th Int'l Conf. Software Testing, Verification and Validation* (ICST 12), 2012, pp. 400–409.
12. Y. Amannejad et al., "A Search-Based Approach for Cost-Effective Software Test Automation Decision Support and an Industrial Case Study," *Proc. Int'l Workshop Regression Testing*, 2014, pp. 302–311.
13. V. Garousi and M.V. Mäntylä, "When and What to Automate in Software Testing: A Multi-vocal Literature Review," to be published in *Information and Software Technology*, 2016; doi:10.1016/j.infsof.2016.04.015.