Test Automation: Not Just for Test Execution

Article *in* IEEE Software · March 2017

DOI: 10.1109/MS.2017.34

2 authors:

Vəhid Gəruslu
Queen's University Belfast
206 PUBLICATIONS **6,429** CITATIONS
SEE PROFILE

Frank Elberzhager
Fraunhofer Institute for Experimental Software Engineering IESE
97 PUBLICATIONS **751** CITATIONS
SEE PROFILE

Test automation: not just for test execution

Vahid Garousi
Software Engineering Research Group
Department of Computer Engineering
Hacettepe University, Ankara, Turkey
vahid.garousi@hacettepe.edu.tr

Frank Elberzhager
Department of Architecture-Centric Engineering
Fraunhofer Institute for Experimental Software Engineering IESE
Kaiserslautern, Germany
frank.elberzhager@iese.fraunhofer.de

INTRODUCTION

Software testing is a costly and effort-intensive activity. According to a study by Pierre Audoin Consultants [1], in 2010 the worldwide costs for testing (including testing software, hardware, and services) amounted to 79 billion euros, and that number was expected to exceed 100 billion euros in 2014. Recent hot topics, such as agile methods, and continuous integration, strongly encourage software engineers to think about how testing can be further automated during the software development, deployment, and delivery process beyond mere test execution.

Software testing is a time consuming activity. It, thus, requires automation (tool support) to reduce costs and ensure high regression.

Different software testing steps can be conducted either manually or in an automated manner. In manual testing, the tester takes over the role of an end-user executing the software under test (SUT) to verify its behavior and find any observable defects. However, in automated testing, using certain test tools, test-code scripts (e.g., using the JUnit framework) are developed and are then executed without human testers' intervention to test the behavior of an SUT. If planned and implemented properly, automated testing could yield various benefits over manual testing, such as repeatability and reduction of test costs (and thus effort). However, if not implemented properly, automated testing will lead to extra costs and effort and could even be less effective than manual testing in detecting faults [2]. While we are aware of companies that already have highly automated their testing activities, many companies, especially small- and medium-sized ones, have great potential to further automate their testing activities.

Automation can be conducted in any phase across the software testing process, not only in test execution.

BRIEF OVERVIEW OF THE STATE OF TEST AUTOMATION

Test automation has a history of over two and half decades, since around 1990 [3], and can lead to many benefits [19]. While test automation mostly started with the automation of test execution, it expanded to other areas and phases of testing too, e.g., test-case design and defect reporting. Large companies such as Microsoft and Google have benefited a lot from automating different test activities, as reported in two recent books ('*How We Test Software at Microsoft*' (2008) [4] and '*How Google Tests Software*' (2012) [5]). In discussions with several test engineers in Turkey, we have also observed that many practitioners practice end-to-end usage of automation from test-case design to execution and to automated defect reporting.

The adoption of test automation, however, varies a lot across different companies; e.g., one company may use automation for automating only some main test cases of a web application, while another company may conduct most of its test activities with automated tools. According to three recent surveys of testing practices from Canada [6, 7] and [8] and the authors' observations based on their involvement in more than 40 testing projects in close partnership with various companies in three countries (Canada, Turkey, and Germany), many test engineers think of automation only in terms of tools for executing test cases, e.g., GUI record/playback tools. Especially small and medium-sized enterprises are often simply unaware of the great potential of automation throughout the entire testing lifecycle, or do not seriously consider the possibility of greater utilization of automation in different test activities.

Most testers think of automation only in terms of tools for executing test cases. They are unaware of test tools for other test activities.

Even Wikipedia seems to have a slightly limited definition of test automation (only covering test execution and evaluation) (as of this writing): "*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*".

As another reference point, the authors reviewed a large number of books on test automation. Aside from textbooks, most of the books on testing describe the best practices in this area. For this purpose, we compiled a list of books published on test automation. In order to keep the workload manageable, we selected books published between 2010 and 2014 in an online spreadsheet http://goo.gl/8aTr4x and checked whether each book only discussed automation of test execution or described other testing activities as well, e.g., test case design. Out of 81 books in this pool, only a handful discussed automation beyond test execution.

Only by being aware of various automated test strategies and tools that can provide assistance in test activities other than test execution would practitioner test engineers be able to fully benefit from these in testing projects and work more efficiently and effectively. The goal of this article is to increase this awareness of test automation beyond just the test execution phase, express further benefits of test automation and to point out that industrial testers should broaden their definition of test automation. The software testing industry in the area of test execution automation is well developed and quite mature, since there are many advanced and powerful tools in this area, e.g., Selenium and various mobile test automation tools such as Testdroid. On the other hand, the state of the practice (industrial usage) regarding the usage of automated techniques for other software testing activities, e.g., automated test-data generation, seems to be quite immature, although there are various tools and techniques in academia aimed at these needs. For example, despite the existence of various prototype tools for search-based software test-data generation (such as EvoSuite [9] and AUSTIN [10]), one wonders why we are only seeing limited penetration (usage) of those tools in industry, e.g., in the automotive sector [10].

To increase the level of automation across all software testing activities and encourage wider usage of the existing test tools for activities other than test automation, more efforts by and collaborations between researchers and practitioners are needed. Let us start with an overview of test automation across the software testing process.

TEST AUTOMATION ACROSS THE SOFTWARE TESTING PROCESS

Typically, a test process consists of several steps from planning to test specification (test-case design), execution, and reporting [11]. To better understand how automation is used during the test process (and not just during execution), we derived six testing activities where a large potential for automation could be seen, focusing on test specification (design and scripting), execution, and analysis (evaluation and reporting):

1. Test-case design: designating a list of test cases or test requirements to satisfy coverage criteria, other engineering goals, or based on human expertise (e.g., exploratory testing).
2. Test scripting: documenting test cases in manual test scripts or automated test code
3. Test execution: running test cases on the software under test (SUT) and recording the results
4. Test evaluation: evaluating the results of testing (pass or fail), also known as test verdict
5. Test-result reporting: reporting test verdicts and defects to developers, e.g., via defect (bug) tracking systems
6. Test management and other test engineering activities: Test management includes activities such as planning, control, monitoring, and effort estimation. Other test activities include test suite minimization and regression test selection.

An overview of these steps is shown in Figure 1.

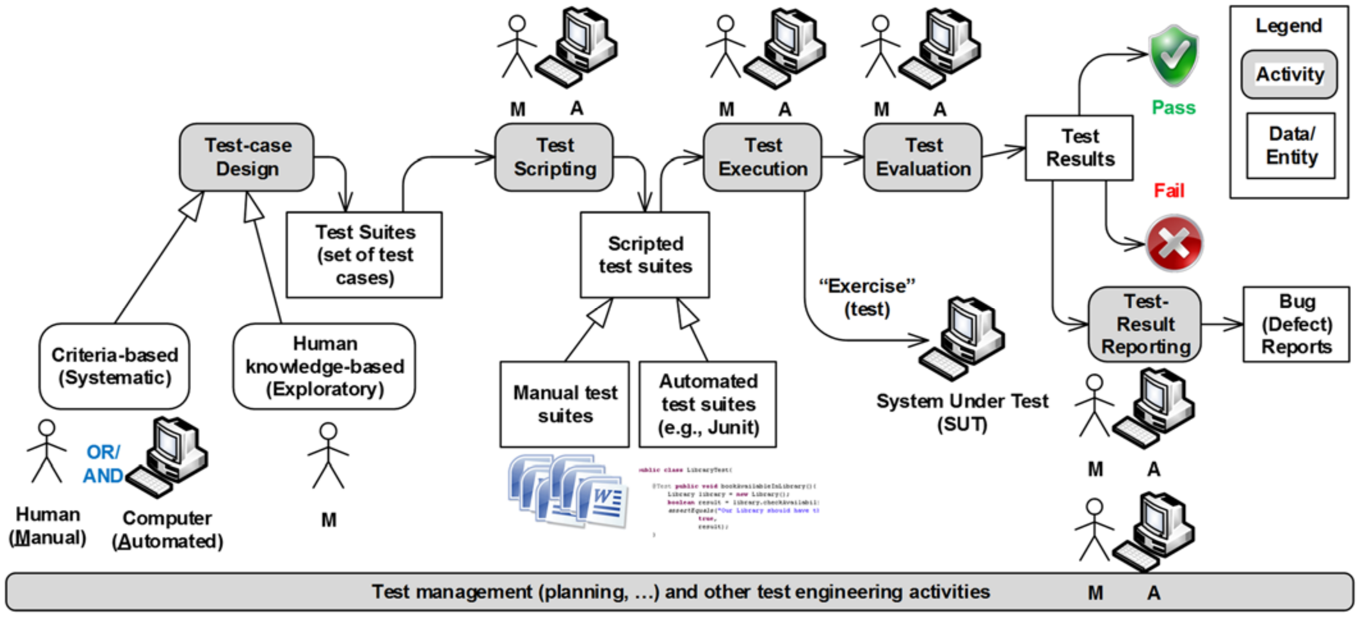


Figure 1- An overview of automation across the software testing process.

A 2013 review paper [12] in the IEEE Software reviewed the different technologies and tools for test automation for test design and execution phases, but did not give a thorough view on the process as we show in Figure 1. Each of the five activities can be done either manually (by a human), in an automated manner (using a software tool), or by mixing the two. This has been shown using labels "*M*" and "*A*" in Figure 1. Also, some activities such as Test-case design can have sub-activities (two types). In the following, we will discuss each of the above six activities and the notion of manual work, partial automation, or full automation in each activity. As can be seen in various sources, such as www.opensourcetesting.org or www.pairwise.org, a vast number of tools exist which serve different purposes, and which can support different testing activities. We picked and review next only a few examples to demonstrate the support for testers. Note that our paper does not intend to provide a full catalogue of tools for each activity, but to raise the importance of automation in different test activities. Full lists of tools can be found in various online sources.

Automation in test-case design

Test-case design is the activity of designating a list of test cases or test requirements. Sub-activities such as identifying test data, including test inputs and expected test outputs, and test paths are part of the test design activity. This activity can be done based either on (1) criteria (e.g., line or requirements coverage)or (2) human knowledge (e.g., exploratory testing) [11].

To conduct criteria-based Test-case design in a fully automated fashion, a large body of knowledge is available, which is often referred to as software test data generation. Test data generation, including inputs and expected outputs, is one of the important parts of the test design activity. Also, for automating combinatorial criteria-based test approaches such as pair-wise testing [11], open-source and commercial tools exist that generate test inputs, such as Hexawise.

Test-case design based on human knowledge is mostly referred to as exploratory testing. As per the nature of this type of testing, it is done fully manually. Criteria-based test-case design can be done manually, in an automated manner, or by mixing the two. In fully manual criteria-based test case design, a tester looks at the requirements or the code (depending on whether black- or white-box testing is done) and derives the test cases without using any tool.

For partial automation, a tester can use any of the many available code coverage tools, but has to expend manual effort to derive test cases in order to increase coverage. The output of the Test-case design activity is a test suite, which is a set of test cases (input and expected output values) or test requirements (e.g., control flow paths to be covered). The next test activity (test scripting) makes use of this output.

Another example tool is called Crawljax, which is an open-source Java tool for automatically crawling and testing Ajax-based web applications. Crawljax explores JavaScript-based Ajax web applications through an even-driven dynamic crawling engine, and automatically creates a state-flow graph of the dynamic states in the Document Object Model (DOM) and the event-based transitions between them. This inferred state-flow graph is then used for automating many types of web analysis and testing techniques, e.g., test-case generation and automated test scripting.

Perhaps one of the best published industrial success stories in utilizing automation in test-case design is that of a Microsoft tool named SpecExplorer, which uses model-based techniques to auto-generate test scripts and which has been empirically evaluated in several studies, e.g., [13].

Automation in test scripting

Test scripting can also be done manually, in a partially automated manner, or in a fully automated way. Manual development of test scripts is obvious, as developing manual test cases is the usual and the basic step that has been done by many manual testers for many years.

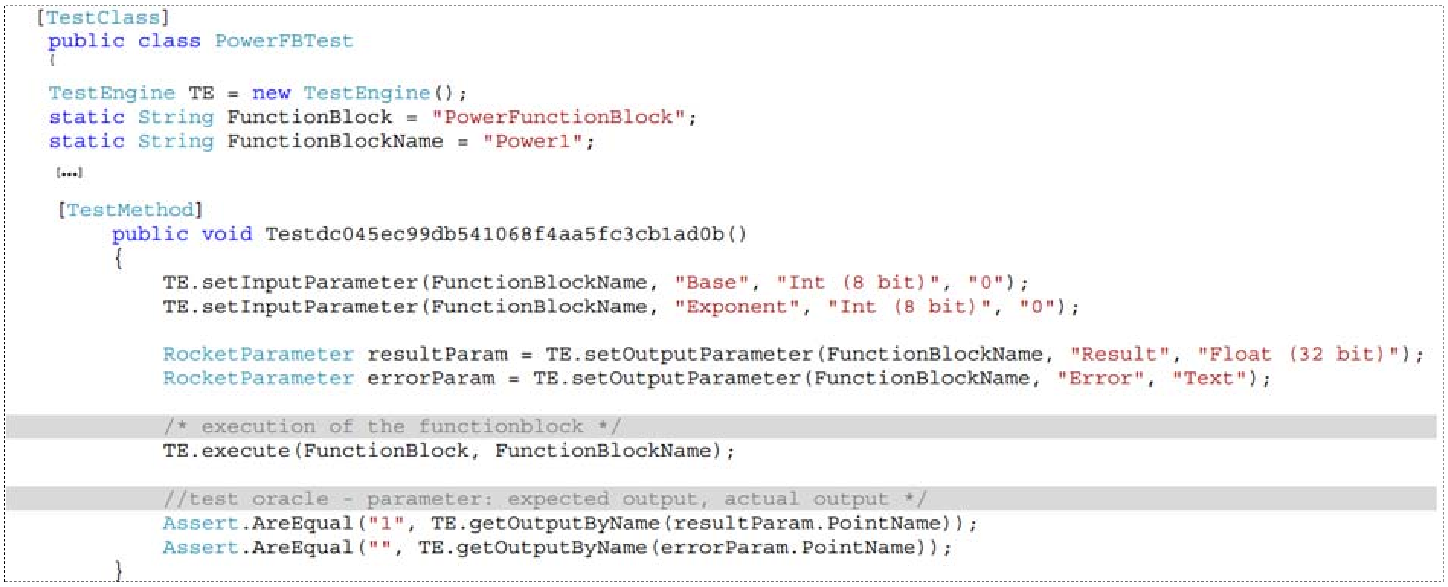
In many contexts, testers no longer have to write test code by hand. There are tools for automated generation of test code.

For partial automation in this context, manual test support tools such as IBM Rational Manual Tester are used to document and store test cases for manual execution later on. Furthermore, there are many tools for testing via Graphical User Interfaces (GUI), which are mostly called "record-and-playback" tools, e.g., HP QuickTest Professional (QTP). Using these tools, testers record a test scenario (test case) by interacting with the SUT's GUI, while the tool automatically records the test log as a test script in the background. When recording (test scripting) is done, the test code can be executed later on as many times as needed without tester intervention. Finally, 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 now available [14-16].

A team from IBM Research India discusses [17] that conventional test automation techniques, such as record-replay and keyword-driven automation, can be time-consuming, can require specialized skills, and can produce fragile scripts. To address these limitations, the team developed a technique and a tool, called TACT (previously "ATA"), for automating the test automation task. ATA uses a novel combination of natural language processing, backtracking exploration, runtime interpretation, and learning to support testers in efficiently creating automated test scripts from manual test cases. Several successful case studies using the tool have been reported, e.g., [17], in various IBM projects. Another tool is EvoSuite which automatically generates JUnit test suites towards satisfying a coverage criterion for Java programs. Microsoft Pex is also another similar tool which automatically generates NUnit test suites.

The outputs of the test scripting activity are either (1) manual test scripts (in a variety of formats) or (2) automated test suites (e.g., using tools such as JUnit, HP QTP). The next test activity (test execution) makes use of test scripts.

In an industry case study, when testing a Supervisory Control and Data Acquisition (SCADA) software, a team of researchers and practitioners developed a customized test tool named AutoBBUT which was capable of automatically generating NUnit test suites (details in [19]). Figure 2 shows an example test script generated by this tool. This approach saved the testers from manually writing about 20K LOC of NUnit test code. In another recent IEEE Software article [16], the first author and his colleague reviewed other tools and test patterns for automation test scripting.



```
[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 example NUnit test script automatically generated by a test tool.

Automation in test execution

Test execution is defined as running the test cases on the SUT and recording the results or observing the SUT's output/behavior [11]. We have observed in our interactions with our industrial partners that when practitioners discuss test automation, they often limit their focus to automation of test execution only.

Similar to the above activities, test execution can also be done manually, in a partially automated manner, or in a fully automated way. We can observe at this point that there are interdependencies among the choices (decisions) made in subsequent activities. For example, if a tester chooses to develop all the tests as automated test suites, test execution will obviously be done in a fully automated manner. In contrast, if the test team decides to develop all their tests as manual test scripts, test execution has to be done fully manually. Test execution is partially automated if one proportion of the test suites are automated scripts and the other are manual test scripts.

A very large number of automated test execution tools exist (e.g., see a 2013 IEEE Software review paper [12] or online lists[1, 2]).

Automation in test evaluation (test oracles)

[1] www.opensourcetesting.org
[2] en.wikipedia.org/wiki/List_of_unit_testing_frameworks

After the SUT is executed by a test case, evaluating the test outcome (pass or fail) is an important phase. There are usually three approaches for this: (1) a human tester may make such a judgment, (2) we may decide to incorporate ("hard-code") test evaluations as verification points (also known as assertions) in the test code, and (3) more advanced techniques may be used to build "intelligent" (learning) test oracles using machine learning and artificial intelligence, e.g., [20]. Approaches (2) and (3) are considered to have automated test evaluation, but with different levels of intelligence. Furthermore, sometimes developers decide to write assertions in production code, which are called code contracts following the design-by-contract paradigm.

Automation in test result reporting

The last phase is usually test result reporting, in which test verdicts and defects are reported to developers for fixing, e.g., via defect (bug) tracking systems. The task of test result reporting is usually performed manually, but there are techniques and frameworks for automated bug reporting, e.g., a .NET library called NBug, which automatically creates and sends bug reports, crash reports with mini-dump, and error/exception reports with stack trace.

Automation in test management and other test engineering activities

Tool support has also been developed for other test activities, such as test set minimization (also includes test redundancy detection), regression test selection, and test repair (conducting co-maintenance on broken automated tests when the SUT has changed). For example, a feature of a test coverage tool named CodeCover is an auto-generated redundancy graph of a JUnit test case that helps testers, in a semi-automated manner, detect redundant tests and conduct test set minimization. In terms of tool support for regression test selection, there are several tools as well, e.g., a commercial tool named xRegress, considers code coverage in selecting the smallest set of regression tests. For test repair, there seem to be tools in academia and not many industrial-strength tools. One of these tools is named ReAssert, which automatically suggests repairs for broken unit tests.

Furthermore, a tool based on genetic algorithms was developed to provide decision-support for the "when to automate?" question [2]. This tool also falls in the category of automation for test management. We used the tool in an industrial setting to find the most optimum set of test case to automate.

EXPERIENCES IN TEST AUTOMATION BEYOND MERE TEST EXECUTION

To provide evidence of success in using automation in test phases (other than execution), we will next discuss the summary of two industrial case studies in which the first author was involved recently.

Case 1: Automation in test scripting using a custom-developed tool

Case #1 is about the automation of test-code scripting using a custom-developed tool for testing a piece of supervisory control and data acquisition (SCADA) software. The system under test (SUT) is called Rocket Monitoring and Control and is developed by a Canadian control software development firm. The system is developed using Microsoft Visual Studio C#. The Rocket SCADA system has a tool called *Automation Engine*, which is an IDE for developing advanced control systems. This tool supports 89 function blocks grouped under 12 categories, e.g., Math, Logic, and Control. For example, one of the function blocks is called *Add* which adds two or more input values and calculates the *Sum* result. The input can be any integer or floating point value, or a text.

There are automated approaches, using machine learning and artificial intelligence, for deciding whether a test has passed or failed.

If we consider three inputs to the "Add Function" unit in the above SUT, and if we apply multi-dimensional category partitioning (without pair-wise testing), we would get 7^3= 343 test cases. Let us assume that automating each of these test cases in NUnit, for example, would require at least 4 Test Lines of Code (TLOC), i.e., one line respectively for each of the xUnit test phases: setup, exercise, verify and tear-down. This would result in 343*4=1,372 TLOC and that is only for one of the 89 units under test. This highlighted the magnitude of the effort needed for writing test scripts manually by test engineers. After systematic investigation and evaluation of the existing automated test-code generation tools, we found that, unfortunately, none of them was applicable to the SUT (details can be found in [19]); so a new tool called AutoBBUT (Automated Black-Box Unit Testing) was developed and utilized empirically in our project [19]. As discussed earlier, AutoBBUT was capable of automatically generating NUnit test suites. Figure 2 showed an example test script generated by this tool. This approach saved the testers from manually writing about 20K LOC of NUnit test code. Cost/benefit measurements of the usage of the tool were conducted by means of precise time logging in the company under study. According to the measurements, we calculated the time savings, Return on Investment (ROI), as a result of using the test tool as follows:

- Benefits (time savings)=87 (initial development of test code without the tool)+87*2 (test code maintenance, if the tool did not exist)= 261 hours
- Costs=120 (development effort for the tool)+3 (test code inspection and completion effort)=123 hours
- Benefits-Costs=261-123=138 hours (net time savings)

Thus, we concluded [19] that the automated test scripting saved a considerable amount of time and effort. Since our collaborative R&D project, the industry partner has started to actively utilize the tool in its daily test activities.

Case 2: Automation in test planning and test-case design using a combinatorial testing tool

In another project, one of our industry partners was facing the 'what to automate?' question [2]. The context was the development and testing of embedded software for the oil and gas industry by a software company based in Calgary, Canada. In an 'action research' project, the team of practitioners and researchers developed a search-based approach (using genetic algorithms) to determine which parts of the system and which test case and test activities to conduct manually or automatically in order to ensure the highest ROI for test automation. The approach is one of the few systematic and measurement-driven decision-support techniques found in the literature. To enable systematic decision-making, we defined a notion called Test Automation Decision Matrix (TADM), an example of which is shown in Table 1. For example, this matrix denotes that test-case design phase for use case UC_1 should be automated.

Table 1-A Test Automation Decision Matrix (TADM) [2]

Use cases	Test-case design	Test scripting	Test execution	Test evaluation
UC_1	1*	0	0	0
UC_2	1	1	1	0
UC_3	1	0	0	1
….	…	…	…	…
UC_n	0	1	1	1

*: 1 = Automated (test activity should be automated), 0 = Manual (test activity should be conducted manually)

To find the most optimal TADM for a given test project, a genetic algorithm was developed and used. Careful measurement of cost and benefits was again performed in the company under study and it was found that, by automating exactly the cases and phases recommended by the decision-support tool, the approach helped the company achieve an ROI of about 341%, without any decrease in test effectiveness (further details in [2]). Furthermore, in that project, we conducted automated test-case design using a commercial pair-wise testing tool called Hexawise. Thus the project included two automation approaches: test planning (answering 'what to automate?') and test-case design.

Test engineers are encouraged to consider adopting automation in all the phases across the software testing process, not only in test execution.

Summary of the two cases

Summarizing the above two case studies, we see that test automation for different test activities (e.g., test-case design, test planning and scripting) can in fact offer a lot of savings for test teams regarding efforts and test cost (based on the quantitative data reported above). Thus, we suggest that test engineers should consider adopting such approaches and tools in their test projects.

CONCLUSIONS AND ROAD AHEAD

A higher degree of test automation beyond mere test execution, as it is often found in industry, can improve the testing process and the resulting quality of the system under test. We presented several test activities, which, if planned and executed properly, would yield good ROI benefits in test automation. Especially with respect to recent and new trends such as agile, continuous integration and delivery, a high degree of automation is needed in order to respond quickly to customer requests, while being able to provide a high level of quality at reasonable costs. Therefore, we encourage especially small and medium-sized enterprises to evaluate their own test automation potential and to take steps towards a higher degree of automation. Of course, every environment has to be clearly analyzed regarding the potential test effectiveness and efficiency improvement if test automation is used, and improvements have to be implemented with the needed rigor. Finally, such experiences should be shared – not only to learn from failed or challenged cases, but, even more so, to convince other companies with success cases. Our empirical studies [2, 19] are examples in those areas.

In both of our industry case studies, we observed major ROI from applying automation in test-case design, scripting and test planning activities.

Automation does not come for free and thus has to be carefully implemented to ensure a successful outcome. For almost all of the above-mentioned test activities, a large set of tools already exists for different environments. However unfortunately, the automated testing tool market is very fragmented in nature, as there exist a lot of different tools for, e.g., different programming languages, environments, or IDEs, which are based on different licensing types (open-source, free or commercial). Planning and implementing a proper test automation 'strategy' in such a fragmented context is necessary, but is not trivial. This is a topic that has recently started to get more attention in the testing industry (e.g., sqta.wordpress.com/test-automation-strategy), and will be one direction of our future work. On the topic of test automation strategy, there is a need to conduct further research and empirical studies in order to be able to answer questions such as:

Planning and implementing a proper test automation 'strategy' is very important.

- How much automation is enough? When and what (test cases) should we automate in the context of a given project?
- Which test phases shall be automated and which ones shall be done manually?
- Which test tool shall be used when and under which conditions?
- Which test cases are better executed manually?
- What prerequisites should be met before we can start implementing test automation?

REFERENCES

[1] Pierre Audoin Consultants (PAC), "Software testing spends to hit Euro 100bn by 2014," *https://www.pac-online.com/software-testing-spends-hit-eur100bn-2014-press-release*, 2014.

[2] Y. Amannejad, V. Garousi, R. Irving, and Z. Sahaf, "A Search-based Approach for Cost-Effective Software Test Automation Decision Support and an Industrial Case Study," in *Proc. of International Workshop on Regression Testing, co-located with the Sixth IEEE International Conference on Software Testing, Verification, and Validation*, 2014, pp. 302-311.

[3] K. C. Archie, O. R. Fonorow, M. C. McGould, R. E. McLear, E. C. Read, E. M. Schaefer, *et al.*, "Test automation system," ed: US Patent #US5021997, 1991.

[4] A. Page and K. Johnston, *How We Test Software at Microsoft*: Microsoft Press, 2008.

[5] J. A. Whittaker, J. Arbon, and J. Carollo, *How Google Tests Software*: Addison-Wesley Professional, 2012.

[6] V. Garousi and J. Zhi, "A Survey of Software Testing Practices in Canada," *Journal of Systems and Software,* vol. 86, pp. 1354-1376, 2013.

[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?," *Journal of Systems and Software,* vol. 83, pp. 2251-2262, 2010.

[8] V. Garousi, A. Coşkunçay, A. B. Can, and O. Demirörs, "A Survey of Software Engineering Practices in Turkey," *Journal of Systems and Software,* vol. 108, pp. 148-177, 2015.

[9] G. Fraser and A. Arcuri, "EvoSuite: automatic test suite generation for object-oriented software," presented at the Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering, Szeged, Hungary, 2011.

[10] 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," in *Search Based Software Engineering (SSBSE), 2010 Second International Symposium on*, 2010, pp. 101-110.

[11] P. Ammann and J. Offutt, *Introduction to Software Testing*: Cambridge University Press, 2008.

[12] M. Polo, P. Reales, M. Piattini, and C. Ebert, "Test Automation," *Software, IEEE,* vol. 30, pp. 84-89, 2013.

[13] W. Grieskamp, N. Kicillof, K. Stobie, and V. Braberman, "Model-based quality assurance of protocol documentation: tools and methodology," *Software Testing, Verification and Reliability,* vol. 21, pp. 55-71, 2011.

[14] S. R. Shahamiri, W. M. N. W. Kadir, and S. Z. Mohd-Hashim, "A Comparative Study on Automated Software Test Oracle Methods," in *International Conference on Software Engineering Advances*, 2009.

[15] D. J. Mosley and B. A. Posey, *Just Enough Software Test Automation*: Prentice Hall Professional, 2002.

[16] V. Garousi and M. Felderer, "Engineering of Software Test-Code: Developing, verifying and maintaining high-quality automated test scripts," *In Press, IEEE Software,* 2016.

[17] S. Thummalapenta, S. Sinha, N. Singhania, and S. Chandra, "Automating test automation," presented at the Proceedings of the 34th International Conference on Software Engineering, Zurich, Switzerland, 2012.

[18] IBM Research India, "TACT: Tools for Automated and Cost-effective Testing," *http://researcher.watson.ibm.com/researcher/view_group_subpage.php?id=3614*, Last accessed: Sept. 2015.

[19] 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," in *IEEE International Conference on Software Testing, Verification and Validation (ICST)*, 2012, pp. 400-409.

[20] K. Frounchi, L. C. Briand, L. Grady, Y. Labiche, and R. Subramanyan, "Automating image segmentation verification and validation by learning test oracles," *Inf. Softw. Technol.,* vol. 53, pp. 1337-1348, 2011.

AUTHOR BIOGRAPHIES

Vahid Garousi is an Associate Professor of Software Engineering in Hacettepe University and also a senior consultant, based in Ankara, Turkey. Prior to that, he worked as an Associate Professor in the University of Calgary, Canada. His research interests include software testing, empirical software engineering and improving industry-academia collaborations in software engineering. He was selected a Distinguished Visitor (speaker) for the IEEE Computer Society's Distinguished Visitors Program (DVP) for the period of 2012-2015. 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. He received a PhD in computer science from the University of Kaiserslautern. Contact him at frank.elberzhager@iese.fraunhofer.de.