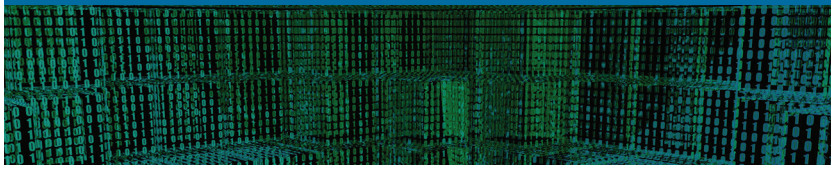
Developing, Verifying, and Maintaining High-Quality Automated Test Scripts

Vahid Garousi, Hacettepe University

Michael Felderer, University of Innsbruck

// Compared to regular source code, test scripts have a life cycle involving different development, quality assessment, and maintenance activities. For such activities to be efficient, we need appropriate techniques, tools, and methods that encompass the test script life cycle. //



AUTOMATED SOFTWARE testing and development of test scripts (test code) are now mainstream in the software industry. For instance, Microsoft test engineers reported "more than a million [automated] test cases written for Microsoft Office 2007."[1]

With the emergence of large, complex automated test suites for commercial and open source software, a great need exists for holistic end-to-end management of test scripts across their life cycle. In a 2013 paper, Vahid Garousi and his colleagues called this process *software test code engineering* (STCE) and presented a summary of the existing tools and techniques for it.[2] Other terms used for STCE are *end-to-end test script engineering* and *test script management*.

In this article, we

- synthesize and report cases from recent STCE projects,
- discuss additional evidence-based viewpoints, and
- summarize Garousi and his colleagues' findings in a more digestible and shorter format targeting practitioners, while highlighting the challenges and raising awareness of STCE.

Review and survey papers have appeared in various areas of software testing—for example, Shin Yoo and Mark Harman's survey on regression-testing minimization, selection, and prioritization.[3] However, almost all those studies are purely academic and don't provide much practical relevance to test practitioners in the context of test scripts. This article aims to fill that gap.

The Emergence of Large, Complex Automated Test Suites

Nowadays, any major commercial or open source software includes automated test suites to verify its functionality. Test automation is particularly prominent in software projects that evolve through many versions because it pays off the most for regression and repetitive testing. For many large-scale systems, automated test suites' size and complexity are constantly increasing. For example, the code base of version 2.1 of the Android OS had 357,933 LOC of JUnit test code, which accounted for

0740-7459/16/$33.00 © 2016 IEEE

17.1 percent of the Java code base of the OS—that is, 2,090,904 LOC.[4] You can easily imagine the major effort needed to develop such a test suite, ensure its integrity (quality), and maintain it alongside the production code (also called the SUT—system under test).

In a Google Tech Talk in November 2005, Jeff Feldstein, then Cisco Systems' manager of software development, gave an example of such growing test code. Referring to the number of automated test suites for a particular router model, he said, "We designed a test system that probably is as complicated as the system itself."[5]

At Google, as an example of a company that stresses test automation, various teams emphasize the importance of careful control and proper management of automated test scripts. According to a test-related webpage of the Chromium projects, "Chromium development places a high premium on [automated] tests, tests, tests, tests, and more tests."[6] It further states that "it is imperative that test suites be updated, maintained, executed, and evolved," showing the importance of these aspects at Google.

Since the 1990s, the software engineering community has been developing rigorous ways to manage automated test suites. This focus led to the concept of STCE.

The Need for STCE

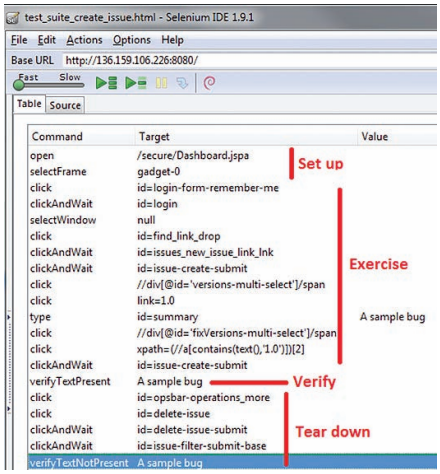
In several recent projects with our industry partners,[7,8] we often observed that although automated test scripts provide many benefits, such as repeatability, predictability, and efficient test execution, script development is tedious and error prone and requires significant up-front investment. Moreover, after the initial development, like any software artifact (such as source code and documentation), scripts require quality assessment and maintenance, and they must coevolve with the production code.

For such activities to be efficient (especially for large-scale test suites), the development of appropriate techniques, tools, and methods that encompass the test script life cycle is essential. This all highlights the importance of proper STCE approaches.

Two examples illustrate the need for a holistic approach to managing the test script life cycle. Figure 1a shows an automated user interface (UI) test case developed using the Selenium tool (www.seleniumhq.org) for testing JIRA, an issue-tracking Web application. This test case verifies JIRA's "enter an issue" feature. The figure highlights the four conceptual steps of testing: set up, exercise, verify, and tear down.

To see how to propagate changes in an SUT to its test code, consider Figure 1a. The test code in Figure 1a initially had /secure/Dashboard.jspa as the starting URL. If that URL changes, the test code won't run and must be revised (comaintained) accordingly.

Figure 1b shows an automated unit test case for the Android platform in JUnit, checking whether the emergency number (911) is properly set. Here, the developers used a *test pattern*,[9] a good practice, to modularize the test code by putting a set of specific verification rules under the function **assertIsValidEmergencyCallerInfo()** and calling it from this example test



(a)

```
/**
 * Checks the caller info. An instance is flagged as an emergency if
 * the number is an emergency number. There's no test for the
 * contact-based constructors because emergency numbers aren't in
 * the contact database.
 */
public void testEmergencyIsProperlySet() throws Exception {
  assertFalse(mInfo.isEmergencyNumber());

  mInfo = CallerInfo.getCallerInfo(mContext, "911");
  assertIsValidEmergencyCallerInfo();

  mInfo = CallerInfo.getCallerInfo(mContext, "tel:911");
  assertIsValidEmergencyCallerInfo();

  mInfo = CallerInfo.getCallerInfo(mContext, "18001234567");
  assertFalse(mInfo.isEmergencyNumber());
}
```

(b)

FIGURE 1. Two example automated test cases. (a) A UI test case developed using the Selenium tool for testing JIRA, an issue-tracking Web application. (b) Unit test code for the Android platform in JUnit, checking whether the emergency number (911) is properly set. These examples illustrate the need for a holistic approach to managing the test script life cycle.

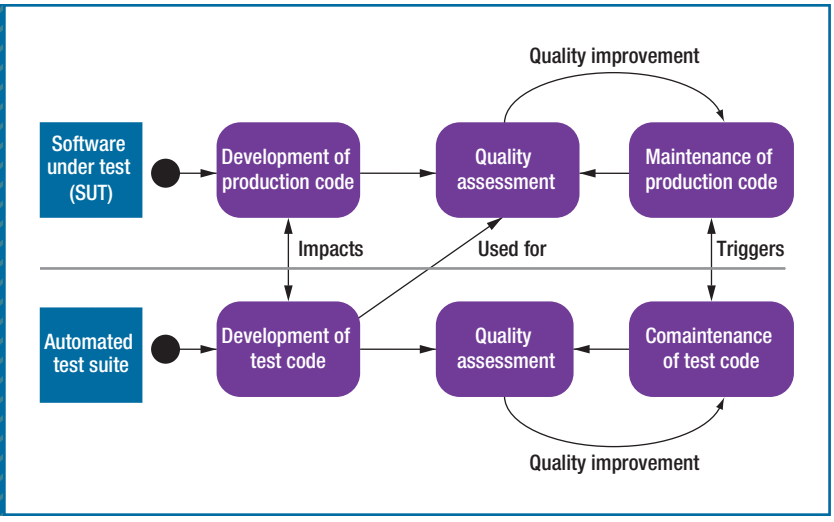


FIGURE 2. An overview of software test code engineering (STCE).[2] The domain at the top is for the system under test (SUT), also called the production code; the one at the bottom is for the automated test suite (the test code).

method. Researchers have observed that using test patterns increases test suite quality.[9] We examine maintenance and the quality assessment of test code in more detail later.

A Closer Look at STCE

STCE comprises two domains; see Figure 2. The one at the top of the figure is for the SUT; the one at the bottom is for automated test code.

STCE has three phases; phase 1 is production code and test code development. In traditional software development processes such as the waterfall model, production code development occurs before test suite development. In recent processes such as test-driven development or test-first development, test code is developed first and subsequently supports production code development. In Figure 2, the bidirectional arrow labelled "Impacts" accounts for these two types of development.

After the first versions of the production code and test code are ready, they undergo phase 2: quality assessment (verification and validation). In phase 3, both types of code are maintained and evolve together. However, maintenance changes are almost always triggered by production code.

A development team might decide to bypass a phase. For example, a team might only develop and maintain the test code as the SUT changes, without improving the test code's quality (for example, through refactoring). However, test code quality is as important as production code quality.

(Throughout the rest of this article, to provide traces to empirical evidence and to enable you to get further details on different topics, we refer to the sources that Garousi and his colleagues reviewed, which are also in the spreadsheet at https://goo.gl/uxbwGd. For example, "source 7" refers to the seventh source in the spreadsheet.)

Test Code Development

Software engineers can manually develop test code or use automated tools (such as Microsoft Pex) to generate it. However, writing high-quality test code isn't trivial,[1] especially for large-scale software.

As an example showing the high effort needed for manual test code development, in a recent industrial project in which Christian Wiederseiner and his colleagues were involved, an estimated 870 hours were spent to manually write automated unit test scripts (in NUnit) for a SCADA (supervisory control and data acquisition) software system (source 7).[7] Motivated by the excessive effort needed to write test code, a tool was developed in-house, and automated test code generation using that tool saved considerable time and effort.

So, similarly to the widely discussed automated code generation approaches, the need exists for fully automated or at least semiautomated test code generation. Various frameworks and tools for this purpose already exist,[8] such as Microsoft Pex, Microsoft SpecExplorer, JML-JUnit, JUB (JUnit test case Builder), TestGen4J, JCrasher, AutoBBUT, and NModel. For example, SpecExplorer uses model-based techniques to automatically generate test scripts and has been empirically evaluated in several studies.[10] However, we encourage practitioners to use such tools and report their experience and findings to increase the empirical body of knowledge[11] by providing guidelines for practical applications.

Quality Assessment and Improvement of Test Code

As with production code, test code quality should be properly verified and improved if necessary. Just as with production code, ensuring test code quality isn't straightforward.

By quality, we mean the test code's correctness in properly testing the production code (functional quality) and whether it's easy to verify and maintain (nonfunctional quality).

Using patterns to ensure quality. Experts have proposed guidelines for ensuring test code quality. For instance, Gerard Meszaros published a large list of test patterns for xUnit test development (source 60).[9] Two such patterns are

- *Delegated Test Method Setup.* Each test creates its own fresh test fixture by calling creation methods from within the test methods.
- *Custom Assertion.* Developers create a purpose-built assertion method that compares only those object attributes that define test-specific equality.

For instance, in Figure 1b, `assertIsValidEmergencyCallerInfo()` is defined as a custom assertion function; in this way, increasing the test code's modularity improves its quality. Meszaros has shared his experience of observing test patterns' benefits in various large-scale test projects.[9]

In several industrial collaborations over the past few years, we've observed that when practitioners didn't use test patterns, they complained about low test code quality, especially during test maintenance. For example, when developing a large automated test suite for testing industrial control software, Wiederseiner and his colleagues developed a tool to automatically generate test scripts (source 7).[7] They employed several patterns—for example, Design for Change (maintainable test code)—and structured test script writing (using set up, exercise, verify, and tear down).

Functional-quality attributes of test code. These attributes check whether a test script properly tests the production code and effectively detects faults. Two important functional-quality issues, which are well known in software-testing theory, apply to automated test scripts (source 33):[12]

- If the test case fails, does the SUT really have a fault?
- If the SUT has a fault, does the test suite detect it?

However, automated test scripts can carry coding-logic-related faults too, owing to human error during their development or maintenance. In keeping with the saying, "Who watches the watchmen?," test engineers should carefully assess and assure scripts' quality. If testers develop test suites that are faulty themselves or can't guard the SUT against potential faults, those test suites will be essentially useless.

So, researchers have developed approaches, tools, and metrics, often based on mutation testing (source 9) and static code analysis (source 31), to assess and verify test suite quality. Many studies have investigated this area. For example, Helmut Neukirchen and his colleagues presented an instantiation of the software quality model proposed by ISO/IEC standard 9126 for test scripts developed in the TTCN-3 (Testing and Test Control Notation) language (source 4).[13] They also developed an approach to assess and improve TTCN-3 test script quality.

Nonfunctional-quality attributes of test code. These attributes involve testing the various test code "-ilities"—for example, maintainability (sources 12 and 32), understandability (source 12), and reliability (source 31). The decayed parts of production code are often called *code smells*—symptoms possibly indicating a deeper problem. In the same way, *test smells* are symptoms whose causes could negatively affect test code quality and make the code difficult to understand or maintain. According to Garousi and his colleagues, as of 2014, 17 studies had proposed and evaluated approaches for detecting test smells.[2]

For example, Stefan Reichhart and his colleagues presented a rule-based approach for detecting 27 types of test smells (source 39).[14] Here are three example types:

- An *eager test* verifies too much functionality. (Figure 3 shows an example of such a test.) It usually has many statements and assertions and is difficult to understand and maintain. Also, more seriously, if the test fails in the middle, the rest of the test logic will be abandoned and won't execute.
- *Conditional logic* is a test containing more than one control flow path, making it less obvious which parts of the test get executed. This increases a test's complexity and maintenance costs.
- An *assertionless test* pretends to assert data and functionality but doesn't. Such a test will always pass.

Negar Koochakzadeh and his colleagues studied in depth a specific type of test smell: *test redundancy* (sources 3, 47, and 49).[15–17] A redundant test case is one whose removal won't affect a test suite's fault detection effectiveness. Redundant test cases can have serious consequences. For example, assume

```
public void testFlightMileage_asKm2() throws Exception {
  // setup fixture
  // exercise constructor
  Flight newFlight = new Flight(validFlightNumber);
  // verify constructed object
  assertEquals(validFlightNumber, newFlight.number);
  assertEquals("", newFlight.airlineCode);
  assertNull(newFlight.airline);
  // setup mileage
  newFlight.setMileage(1122);
  // exercise mileage translator
  int actualKilometres = newFlight.getMileageAsKm();
  // verify results
  int expectedKilometres = 1810;
  assertEquals(expectedKilometres, actualKilometres);
  // now try it with a canceled flight:
  newFlight.cancel();
  try {
    newFlight.getMileageAsKm();
    fail("Expected exception");
  } catch(InvalidRequestException e) {
    assertEquals("Cannot get cancelled flight mileage",
    e.getMessage());
  }
}
```

FIGURE 3. An example of an eager test, adapted from www.xunit.patterns.com. Such a test verifies too much functionality. It usually has many statements and assertions and is difficult to understand and maintain.

that two test cases test the same feature of a unit. If one of them is updated correctly with the unit but the other one isn't, the test suite's integrity will be questionable. That is, one test case will fail while the other passes, leaving the quality assurance team in an ambiguous situation. Koochakzadeh and Garousi developed a tool to automatically detect test redundancy (source 47).[16] An empirical analysis of four well-known open-source test suites showed high redundancy, demonstrating the need for such tools and more awareness about test redundancy in the test automation community.

Comaintenance

Whenever the SUT changes, its test suites will usually be affected and will need to be changed. This is, in fact, test-code comaintenance: adaptive maintenance of test code as the production code is being maintained. According to Garousi and his colleagues, as of 2014, more than 23 studies had investigated this subject.[2]

Yuri Shewchuck and Garousi reported their experience maintaining a functional GUI test suite developed using the IBM Rational Functional Tester (source 19).[18] The SUT was jEdit, an open source text editor. Shewchuck and Garousi first developed a test suite consisting of 71 test cases for jEdit 3.2.2. When executing the GUI test suite on jEdit 4.0 and 4.1, they counted the test cases that passed or failed. In addition, they counted the number and types of changes to the failed test cases that were necessary owing to code or GUI changes in the SUT. Figure 4 shows that information. They then measured the effort needed to maintain the test suite and the types and scale of test maintenance activities for jEdit 4.0 and 4.1. For example, they counted the test cases that needed rerecording entirely from scratch, owing to SUT changes.

Of the 71 test cases, only 19 (27 percent) passed in version 4.0 and 27 (38 percent) passed in 4.1. More test cases passed for version 4.1 owing to the types of SUT changes in the two versions. Of the failing test cases (52 in version 4.0 and 44 in version 4.1), 10 (14 percent) in each of the two versions detected a defect, as per the root cause analysis. However, the remaining 42 (59 percent) and 34 (48 percent) test cases failed not because of an SUT defect but because they were broken.

The repair of those failed test cases involved two types of test maintenance:

- Updating test oracles. For example, the test oracle had to be revised when the testing moved from one version to the next.
- Rerecording all or a portion of a test case. For example, the new version no longer contained a specific text edit box in its GUI.

For version 4.0, 24 test cases (34 percent) needed an oracle update; the other 18 (25 percent) needed rerecording. For version 4.1, 10 test cases (14 percent) needed an update; the other 24 (34 percent) needed rerecording.

This example shows that test engineers will likely face situations in which automated test cases developed for one version of an SUT produce different results on the next version. Test engineers must carefully assess each failing test case and answer this question: Has the test case really detected a defect or just produced a false positive (a test case that must be repaired for the next SUT version)? Although some research has dealt with this issue, we need more techniques and tool support to reach more mature, efficient STCE.

From our practical experience automating unit and system (GUI) testing (and from the example test code in Figure 2), we can see how these

two test types are similar and different. For example, they both do some type of setup, exercise the SUT, and then verify the functionality. However, they test the SUT through different interfaces. So, for each type of testing (unit versus system testing), the STCE-related challenges and techniques to address them differ. For example, how should you develop the GUI test code (for instance, by using the tool Selenium) to minimize the necessary maintenance efforts?

Some test antipatterns relate to comaintenance. For example, *fragile test* refers to a test not compiling or running when the SUT changes in ways that don't affect the part the test is exercising. As Meszaros pointed out, fragile tests increase test maintenance costs by forcing testers to visit many tests each time they modify a system's functionality, which is particularly critical on projects with highly incremental delivery.[9] He suggested looking for patterns of how the tests fail and then asking "What do these broken tests have in common?" This should help you understand how the tests are coupled to the SUT. You should then look for ways to minimize this coupling. Figure 5 summarizes the process of determining test case sensitivity.

Moving Forward

Here are four recommendations for the path forward in this area.

First, both researchers and practitioners need more awareness of the entire test script life cycle and STCE. Such awareness will likely encourage further industry–academia collaboration.

Second, we need more empirical evidence and studies on all STCE-related activities, to answer these questions:

- How well do the existing tools help testers perform automated or semiautomated generation of test code, and what cost savings do they provide?
- How useful are the existing test patterns? How do different patterns affect test code quality?
- How can we characterize test code quality, and which software quality attributes are relevant for test code? How can we detect and deal with test code quality problems—that is, test smells?
- How can we systematically and efficiently conduct comaintenance and coevolution of production code and test code, while conserving test code quality?
- How does production code maintenance affect test code?
- How do comaintenance and coevolution of unit test suites compare to comaintenance and coevolution of system test suites?

Third, we need additional test patterns that help test engineers write high-quality test scripts. We also need general guidelines for all STCE-related activities, taking empirical results into account.

Finally, although many tools assist various STCE activities (Garousi and his colleagues identified 41 tools;[2] see the details at https://goo.gl/uxbwGd), the need exists to evaluate those tools in large projects. We also need to identify the effectiveness and weaknesses of those tools and pinpoint new areas requiring tool support.

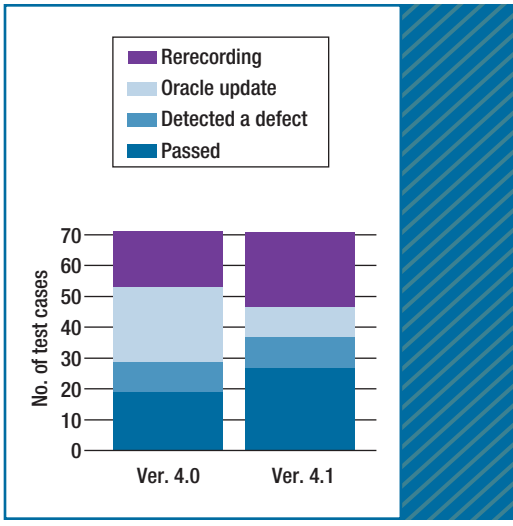


FIGURE 4. Types of required test maintenance activities related to a functional system GUI test suite developed using the IBM Rational Functional Tester.[18] This example shows that test engineers will likely face situations in which automated test cases developed for one version of an SUT will produce different results when executed on the next version.

Test code engineering needs the same level of attention that production code engineering has received. To manage current and future challenges in this area, the software engineering community should address the recommendations we pointed out in this article. We invite all the practitioners who develop test code using any test tool or framework to review the wealth of knowledge and methods we've discussed and use them for developing, verifying, and maintaining high-quality automated test scripts in their test automation projects.

References

1. A. Page, K. Johnston, and B. Rollison, *How We Test Software at Microsoft*, Microsoft Press, 2008.
2. V. Garousi, Y. Amannejad, and A. Betin-Can, "Software Test Code Engineering: A Systematic Mapping," *J. Information and Software Technology*, Feb. 2015, pp. 123–147.

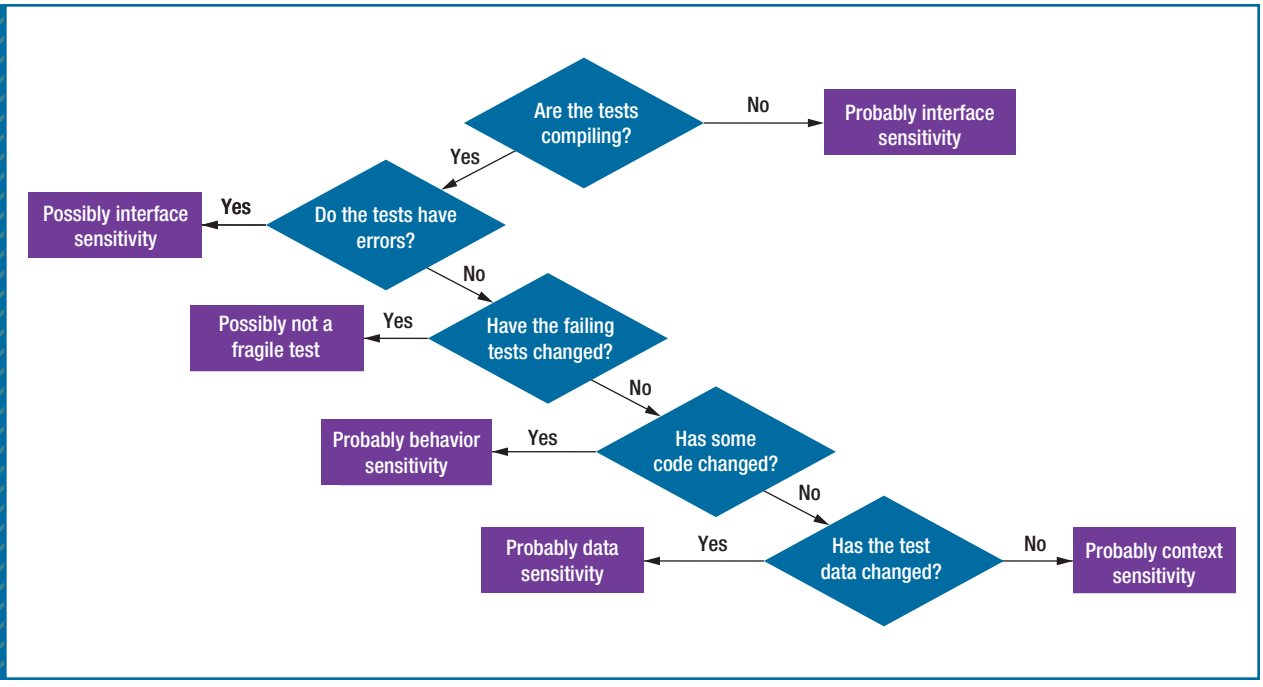


FIGURE 5. Trouble-shooting fragile tests, which don't compile or run when the SUT changes in ways that don't affect the part the test is exercising.[9] Looking for patterns of how the tests fail should help you understand how the tests are coupled to the SUT.

3. S. Yoo and M. Harman, "Regression Testing Minimization, Selection and Prioritization: A Survey," *Software Testing, Verification and Reliability*, vol. 22, no. 2, 2012, pp. 67–120.
4. V. Garousi, R. Kotchorek, and M. Smith, "Test Cost-Effectiveness and Defect Density: A Case Study on the Android Platform," *Advances in Computers*, vol. 89, 2013, pp. 163–206.
5. J. Feldstein, "How to Recruit, Motivate, and Energize Superior Test," Google Tech Talks, 2005; www.youtube.com/watch?v=PyhtoQz7RHY.
6. "Testing and Infrastructure," Chromium Projects; www.chromium.org/developers/testing.
7. C. Wiederseiner et al., "An Open-Source Tool for Automated Generation of Black-Box xUnit Test Code and Its Industrial Evaluation," *Testing—Practice and Research Techniques*, LNCS 6303, Springer, 2010, pp. 118–128.
8. 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 6th Int'l Conf. Software Testing, Verification and Validation* (ICST 12), 2012, pp. 400–409.
9. G. Meszaros, *xUnit Test Patterns*, Pearson Education, 2007; http://xunitpatterns.com.
10. 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.
11. J.S. Kracht, J.Z. Petrovic, and K.R. Walcott-Justice, "Empirically Evaluating the Quality of Automatically Generated and Manually Written Test Suites," *Proc. 14th Int'l Conf. Quality Software* (QSIC 14), 2014, pp. 256–265.
12. B. Daniel et al., "ReAssert: Suggesting Repairs for Broken Unit Tests," *Proc. 24th IEEE/ACM Conf. Automated Software Eng.*, 2009, pp. 433–444.
13. H. Neukirchen, B. Zeiss, and J. Grabowski, "An Approach to Quality Engineering of TTCN-3 Test Specifications," *Int'l J. Software Tools for Technology Transfer*, vol. 10, no. 4, 2008, pp. 309–326.
14. S. Reichhart, T. Gîrba, and S. Ducasse, "Rule-Based Assessment of Test Quality," *J. Object Technology*, vol. 6, no. 9, 2007, pp. 231–251.
15. N. Koochakzadeh and V. Garousi, "A Tester-Assisted Methodology for Test Redundancy Detection," *Advances in Software Eng.*, vol. 2010, 2010, article 6.
16. N. Koochakzadeh and V. Garousi, "TeCReVis: A Tool for Test Coverage and Test Redundancy Visualization," *Testing—Practice and Research Tech-*

niques, LNCS 6303, Springer, 2010, pp. 129–136.
17. N. Koochakzadeh, V. Garousi, and F. Maurer, "Test Redundancy Measurement Based on Coverage Information: Evaluations and Lessons Learned," *Proc. 2009 Int'l Conf. Software Testing Verification and Validation* (ICST 09), 2009, pp. 220–229.
18. Y. Shewchuk and V. Garousi, "Experience with Maintenance of a Functional GUI Test Suite Using IBM Rational Functional Tester," *Proc. Int'l Conf. Software Eng. and Knowledge Eng.*, 2010, pp. 489–494.

ABOUT THE AUTHORS

VAHID GAROUSI is an associate professor of software engineering at Hacettepe University. 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.

MICHAEL FELDERER is a senior researcher in the University of Innsbruck's Institute of Computer Science. His research interests include software and security testing, software engineering processes, requirements engineering, empirical software engineering, and industry–academia collaboration. He also transfers his research results into practice as a consultant. Felderer received a PhD in computer science from the University of Innsbruck. Contact him at michael.felderer@uibk.ac.at.