Editor: **Christof Ebert**
Vector Consulting Services
christof.ebert@vector.com

Testing Software Systems

Christof Ebert, Divith Bajaj, and Michael Weyrich

From the Editor

Testing is often considered boring. Wrong! Testing matters for many reasons. Not only is it the area where most of the money goes across the product life cycle but new challenges, such as adaptive software, agile deployment, and artificial intelligence (AI)-driven systems, demand novel test technologies. Michael Weyrich, Divith Bajaj, and I evaluate current technologies for software testing, providing an overview of methods and tools, including systematic risk assessment for each test activity. A case study for testing AI systems provides insight into novel intelligent testing strategies.—*Christof Ebert*

SOFTWARE IS CONSUMING the world—and defects are consuming the software. While we entrust our entire society to software, we rarely think about the underlying quality. Yet quality, and especially liability, matters. The number of product recalls and amount of legal action due to insufficient software quality is fast increasing. As software engineers, we, of course, know that quality must be designed into a product, not tested. But as human beings, we also know that defects happen. There is no way around systematic verification and validation (V&V).

Digital Object Identifier 10.1109/MS.2022.3166755
Date of current version: 20 June 2022

Today, organizations invest more than one-third of their IT budgets in quality assurance and testing and more than half of their life cycle budgets in safety-critical systems[1–4] for their processes and products. With more agility and continuous updates, the number of testing activities is increasing. It implies, for instance, continuous verification and continuous validation, which go way beyond classic smoke tests. There is a need for automation through the continuous-X pipeline and even more so for entirely novel test schemes, such as cognitive testing (CT) in autonomous systems.

Why Is Testing So Difficult?

Software-driven risks and problems are steadily increasing, as we see with the fast growth of defects, cybersecurity attacks, and insufficient usability. Testing must therefore evolve even faster. At the same time, we need to prepare on a dual track for enhanced quality. Test strategies not only mean finding defects but also hardening systems for robustness. Corrections and changes must be deployed in a fluid scheme, reliably over the air. Resilience strategies, such as graceful degradation and fail-operational scenarios, need to be designed and deployed.

Systematic continuous V&V is the call of the day as a result of continuous integration (CI)/continuous delivery (CD) processes. Yet, it is are rarely achieved in practice for the following reasons:[1–5]

0740-7459/22©2022IEEE

- State explosion with continuous updates and flexible DevOps distribution prohibit classic V&V methods, such as functional testing and code coverage.
- The strength of defined software processes has eroded under an agile "no process" attitude, as we see in the increasing number of interface and integration failures.
- Continuous X means that software deployed yesterday at a given quality level will be dated today, with much lower quality.
- Defects and cybersecurity vulnerabilities grow with ever-higher system complexity and increasing time pressure.
- Cybercrime and attacks are deployed by entire countries and for-profit organizations, totaling more volume than global drug trafficking.
- Machine learning (ML), artificial intelligence (AI), and adaptive platforms continuously change underlying software and thus prohibit traditional coverage schemes and functional testing.
- The widening gap between complexity and available competence and capacity means that an increasing amount of software is developed by unskilled people.
- The traditional quality attitude is diminishing, with a new generation of developers that grew up believing there is an undo button for every action.

Mitigation is far from trivial, considering the evolution with highly distributed services, automatic functions of autonomous systems and increasingly complex interactions with the real world. This raises many questions about the validation of autonomous vehicle systems. With AI and ML, we need to satisfy algorithmic transparency. For instance, what are the rules in an obviously not-anymore-algorithmically tangible neural network to determine who gets credit or how an autonomous vehicle might react when facing several hazards at the same time? Classic traceability and regression testing certainly will not work. Rather, future V&V tools will include more intelligence based on big data exploits, business intelligence, and systems' own learning to improve software quality in a dynamic way.[5]

> There is a need for automation through the continuous-X pipeline and even more so for entirely novel test schemes, such as cognitive testing in autonomous systems.

Systematic testing must take a variety of perspectives,[2,3,5] including the following:

- What is the specified functionality?
- Does the system follow the specification?
- Are all relevant and critical situations and their correlations adequately specified?
- What is the intended functionality, and is it safe?
- How do we trace decision making and make judgments about it? How do we supervise this?
- How do we define reliability in the event of failure?
- What if the intended functionality fails by accident, improper design, or attacks?
- What negative outcome must be prohibited in any case?
- What adaptive or learning subsystem will need which type of regression testing?
- Which information about software updates and ML must be made available to users?
- How do we identify test end criteria in continuous development cycles?
- What are the minimum viable test strategies to be executed during changes, updates, and learning in AI-based systems?

These questions must be addressed with the right technology. For legal reasons, original equipment manufacturers must demonstrate that products are sufficiently tested.

Test Technologies

Testing has shifted from the brute force and ad hoc approach of previous decades. Methods such as test-driven requirements engineering and test-driven development (TDD) are used to start testing before design.[4] A good basis for achieving software quality is the ISO/International Electrotechnical Commission (IEC) 12207 standard and its software

Software-driven risks and problems are steadily increasing, as we see with the fast growth of defects, cybersecurity attacks, and insufficient usability.

V&V processes.[6] It uses V-shaped abstraction, where for each process on the left (development), there is a corresponding V&V process on the right (testing). Based on this, we can distinguish three areas connected by traceability, namely, requirements for design and for validation. ISO/IEC/IEEE 29119 more precisely specifies various software test activities.[7] Figure 1 presents the three areas, often labeled *triple peaks* as a play on the name of a TV series from last century. In this abstraction, the unit tests verify whether the source code complies with the low-level design. The integration tests verify whether previously tested components fit, that is, whether they work in an integrated manner. The system tests check whether the fully integrated product meets the specifications. And finally, the acceptance tests verify whether the product meets the expectations of the user or client.

The ISO 25000 series on systems and software engineering defines some of the main product quality characteristics and thus enables measurement and test quality requirements[5] as follows:

- *Functional suitability*: the degree to which a product or system provides functions that meet stated and implied needs when used under specified conditions
- *Security*: the degree to which a product or system protects information so that people and other products and systems have appropriate data access
- *Maintainability*: the effectiveness and efficiency with which a product or system can be modified for improvement, correction, and adaptation to changes in environments and requirements
- *Usability*: the degree to which a product or system can be employed by specified users to achieve prescribed goals with effectiveness, efficiency, and satisfaction
- *Performance efficiency*: the performance relative to the number of resources used under stated conditions.

Table 1 provides an overview of test activities mapped to methodologies,

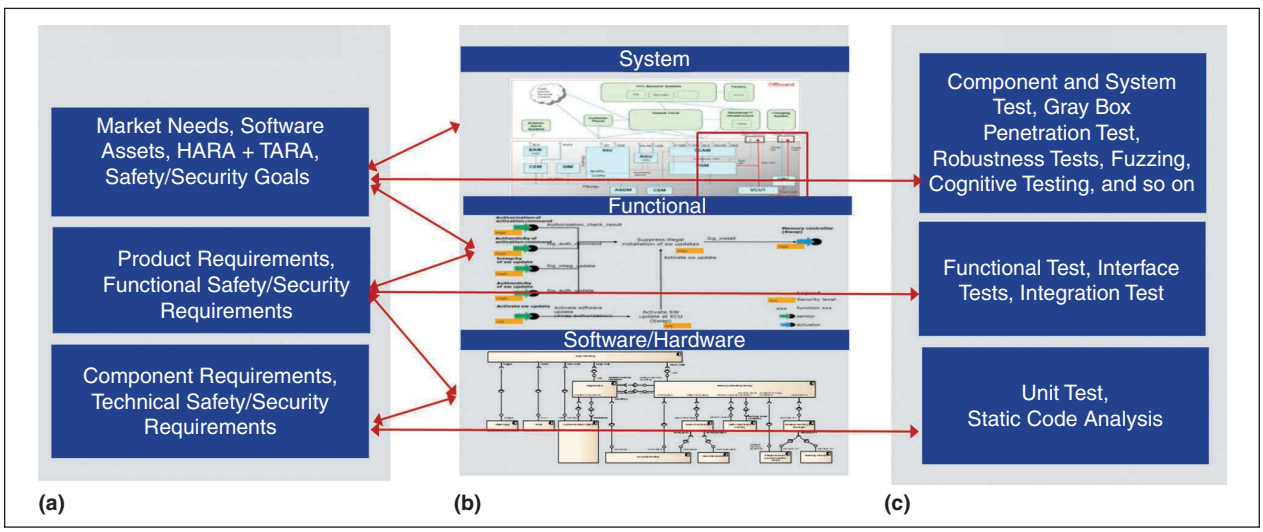

FIGURE 1. The triple-peak model (vertical tiers) and its different abstractions (horizontal tiers). The (a) requirements, (b) design, and (c) tests. HARA: hazard assessment and risk analysis; TARA: threat assessment and remediation analysis.

Table 1. The test activities, with methods, tools, and usage schemes.

Activity	Test methodology	Defect types	Blind spots	Tools	Effort
Static analysis	• Method to detect defects at early stage development of software unit • Design inspection for software unit • Review of end user functionality, e.g., use cases • Validation of functional requirements • Review of descriptions for database, interfaces, input/output, hardware, software and network • Techniques: code reviews, walk-throughs, inspections, and flow analysis	• Design defects • Insufficient functional requirements • Inconsistent design and architecture specifications • Memory and buffer overflows • Code complexity • Code, syntax, and standardized labeling errors	• Dynamic behaviors of the code execution • Processor utilization • The number of reported deficiencies makes most engineers turn off warnings, thus creating massive blind spots	Lint, Soot, CheckStyle, SourceMeter, KlocWork, Parasoft C/C++ with CERT C rules support, CodeSonar C/C++, Imagix 4D, Testwell CMT++ CTC++, VectorCast	Medium, slightly automated
Unit testing	• Method to test a software unit • White-box testing • Observes TDD • Tests to verify functional (software) requirement “input X generates Y” • Techniques: equivalence classes analysis for high-variance classes, boundary value analysis, code coverage testing (statement, branch and modified condition/decision coverage)	• Potential bugs in the code base • Missing functionality in comparison to the requirements • Buffer overflows • Exception control and process utilization	• Integration errors • System errors • Functionality across multiple software units • Intrinsic dependencies among software units • Quality requirements • Unclear coverage schemes	Junit, PHPUnit, NUnit, UnitTest ++, pytest, , Testwell CMT++ and CTC++, CANTATA, ADATest 95, Mockito, TestNG, VectorCast, CANoe4SW	Low, highly automated
Function testing (software)	• Method to test SW functionality based on System requirements • Black-Box Testing of SW Application • Analysis of defined, expected, and recorded input and output values • Test Case definition based on Functional and Business Requirements • Techniques: Smoke Testing, Sanity Testing, Interface Testing, Acceptance Testing	• SW Application’s main functions • Functionalities of the user interface, APIs, I/O’s and SW Database • Operating Limitations in proprietary application specific SW Units • Functionality spanned across multiple SW units	• Quality Requirements • Hardware deficiency cannot be captured • Code based vulnerabilities cannot be identified • AI and adaptive systems are not covered	TestComplete, UFT, Parasoft Suite, Selenium, QTP, SoapUI, Watir, Wireshark, Tcpdump, LoadRunner, JMeter, VectorCast, VT System	Medium, highly automated

(*Continued*)

Table 1. The test activities, with methods, tools, and usage schemes. (cont.)

Activity	Test methodology	Defect types	Blind spots	Tools	Effort
Function testing (component)	• Method to test software component, based on system requirements • Black-box testing of software application • Tests a slice of functionality of the whole system • Validation against test requirements and use cases • Analogous to static analysis and unit testing	• Errors in functions (stubs) and calling functions (drivers) • Design defects • Errors in software component tasks • Bugs and inaccuracies in the test plan and specifications	• Quality requirements • Design conditions • Design preferences • Structural limitations of the component under test	CANalyzer, CANoe, vTESTstudio, VectorCAST, JUnit, PHPUnit, NUnit, UnitTest ++, Pytest Selenium, QTP, SoapUI, DSO, software-defined radio, Locust, NeoLoad	Medium, highly automated
Integration test	• Method to test a software application i.e., the combination of multiple software units • Approach: White box, black box, and gray box • Techniques: Big bang, incremental, top down, and bottom up • Priority testing of critical modules (top-down) • Integration testing supports evaluation of the software application against requirement changes	• Integration errors from different software units • Functionality across multiple software units • Intrinsic dependencies between software units • Faults at external interfaces and input/output ports • Design flaws can be identified • Quality requirements	• Source of specific defect • Code-based vulnerabilities • Operating limitations in proprietary application-specific software units • Interface links could be missed as number of interfaces increases with number of software units	Cantata, LDRA TBrun, LDRAunit, Citrus, Protractor, Shodan, FitNesse, Hamcrest	High, highly automated
Embedded testing with software in the loop, model in the loop, and hardware in the loop	• Software in the loop for software functionality by simulating the system and hardware • Model in the loop for testing software in a model-based environment • Hardware in the loop by including actual (embedded) components • Validation of different test scenarios by examining the expected and recorded behavior of the input and outputs • Validation of software requirements • With hardware in the loop, real-time simulations are possible	• Functional errors of a software application • Coding errors from implicit code coverage testing (exit criteria for software-in-the-loop testing) • Functional errors of the hardware/device under test as it is “in-the-loop” simulated • Model in the loop: bugs and inaccuracies of the modeled hardware and software • Hardware in the loop: defects of interfaces, exchange messages, and networks • Dependencies between software and hardware	• Source of specific defect • Code-based vulnerabilities • Operating limitations in proprietary application-specific software/hardware units	Software in the loop: CANoe, MATLAB, SimuLink, VSim, RT-LAB, vVIRTUALtarget, IoTIFY, IBM Bluemix (now IBM Cloud), TestView, ScopeView. Model in the loop: PREEvision, MATLAB, SimuLink, vVIRTUALtarget, VISUALCONNX, MICROGen, LOOPY, PLECS, Dymola, ASCET, Scilab Hardware in the loop: VT System, PixHawk, vTESTstudio, DSpace, CANoe, NI Lab View, VeriStand, PXI, CompactRIO, LabVIEW, TestStand	High, highly automated

(*Continued*)

Table 1. The test activities, with methods, tools, and usage schemes. (cont.)

Activity	Test methodology	Defect types	Blind spots	Tools	Effort
System test	• Method to validate a software product, i.e., the integration of single or multiple applications • Validation against system requirements • Black-box testing • Derivation of end-to-end testing scenarios for components of software application • Dedicated volume testing, e.g., stress, performance, load • Test for quality requirements, such as availability, accessibility, usability, and so on	• System errors from different software applications • Functional defects of a software product • Defects of communication interfaces, exchange messages, and network functionality • Deficiencies of input/output interfaces • Functionality across multiple software applications • Intrinsic dependencies among software applications • Design flaws can be identified	• Source of specific defect • Code-based vulnerabilities • Operating limitations of proprietary application-specific software applications • AI and adaptive systems are not covered	Quality Center/ALM, SpiraTest, TestMonitor, Eggplant, vFlash, vMDM, vConnect, Postman, Selenium, Wireshark, SoapUI, TestingWhiz, TestComplete, Ranorex	Very high, moderately automated
Security testing	• Technique to validate system and security goals • Fuzzing of the electronic control unit bus through signal and frame fuzzing • Validation of security techniques and testing for robustness • White box, black box, and gray box (interface discovery, network discovery, network penetration test, and software penetration test) • Types: signal and frame fuzzing • Combinatorics: sequential, pairwise, combinatorial, and endless	• Vulnerabilities of the SW and system design, e.g., integrity, confidentiality • Communication-based vulnerabilities • Debug interface-based vulnerabilities • Unintended behavior (spoofing identity, tampering with data, repudiation threats, information disclosure, denial of service, and elevation of privileges) • Examination of hidden pathways that may lead to exploiting a vulnerability	• Requirement deficiency cannot be captured • Hardware deficiency cannot be captured • Code-based vulnerabilities cannot be identified • Brute force fuzzing and performance restrictions result in missed defects and vulnerabilities	VectorCAST, CANoe, vTESTStudio, Kali Linux, hardware/software debuggers, controller area network/local interconnect network/Ethernet/ FlexRay Interface	Very high, only partially automated for fuzzing
Services testing	• Validate webservices and service-oriented architecture; similar to unit test but on different abstraction levels • Webservice implementation approaches: Simple Object Access Protocol, Representational State Transfer, and Web Services Description Language • Software applications employing webservices for data communication	• Faults in the protocol used to create a webservice • Missing and malformed parameters • Regulation address mismatches	• Webservice availability errors • Interface errors • Immutability of existing and new webservices interface to client applications	SoapUI, Axis2 API, SOAPSonar, SOAtest, TestMaker, Postman, Httpmaster, vREST	High, can be well automated

(*Continued*)

Table 1. The test activities, with methods, tools, and usage schemes. (cont.)

Activity	Test methodology	Defect types	Blind spots	Tools	Effort
Cloud testing	• Testing of software applications, software units, and systems by using cloud services. • Software as a service testing is accompanied with testing as a service • Quality assurance testing via cloud tools • Testing cloud tools, architecture, infrastructure, platforms, and so on • Test of cloud services and connectivity • Performance • Cybersecurity • Robustness	• System errors from different software applications • Functional defects of a software product • Defects of communication interfaces, exchange messages, and network functionality • Deficiencies of input/output interfaces • Configuration management helps avoid data loss and data recovery	• Synchronization faults among cloud models • Security and privacy concerns for over-the-air and cloud implementations • Server, storage and network failures impacting testing • Connectivity and availability concerns	SOASTA CloudTest, LoadStorm, BlazeMeter, Amazon Web Services, Jenkins, Xamarin Test Cloud, Nessus, Gatting, Apache JMeter, Selenium	High, automation for service access and performance testing
CT	• Validation of system via cognitive test methods • Black-box testing • Methods: AI/ML algorithms, mathematical models • Employing learning to remove redundancy in test case generation (brute force) • Software application accounted for undiscovered scenarios • Accounting software application for corner, misuse, and abuse cases • Scenario-based testing • Works with X-in-the-loop test suite	• Functional defects of a software application in different scenes, situations, and scenarios • Complements defects found in functional, integration, and system testing • Real-time faults detected if approach is used with X-in-the-loop test suite	• Quality requirements • Software fault localization difficult • Hardware fault localization difficult	Requirements engineering databases, such as DOORS, TensorFlow, Apollo Baidu, Autoware, Open pilot, MATLAB, LGSVL, Vires, dSPACE ASM Toolchain, Carla, CarSim, and AirSim	High, highly automated
Qualification testing	• Test product against relevant stakeholder, functional, and business requirements • Result-driven method to ensure fidelity of software to requirements • Pass/fail criteria for qualification assessment • Quality requirements also included • Failure model and effects analysis for qualification • Test of software against contractual qualification levels, e.g., similarity, comparison, goal certification, and life cycle prediction	• Performance features of software against requirements checklist • Potential defects that may lead to failure in qualifying software/hardware • Vulnerabilities of software against stress testing	• Design flaws • Design conditions • Design preferences • Structural limitations • Source of specific defect • Code-based vulnerabilities • Operating limitations of proprietary application-specific software applications	Requirements engineering databases, such as DOORS, PREEvision, PCAN View, PCAN explorer, CS+, S32 Studio, and LT Spice	High, not automated

typical defect types, risks, tools, and relative effort. The structure is based on current standardization and industry practice.[2,5,8] We enriched the entries with our experience working for three decades in testing and consulting.

> Cybercrime and attacks are deployed by entire countries and for-profit organizations, totaling more volume than global drug trafficking.

Novel Test Technologies

Testing is at a change point both in research and practice. Traditional methods do not suffice for autonomous systems, which have issues related to state explosion, trustworthiness, and safety, and IT security. Safety of the intended functionality (or SOTIF as it is called) examines the approval of safety-critical autonomous systems. A regressive validation is a test that, after changing the control algorithms, performs a new check and ensures that a function gains more importance with the referenced CI/CD pipeline, which frequently updates software.

We will face future scenarios where software-defined systems and maybe whole infrastructures must not be begun if they do not include the latest software upgrades, which themselves must be thoroughly tested. Safety-critical systems, such as automotive vehicles, medical devices, aerospace vehicles, and manufacturing facilities, fall into this category, as they are increasingly defined by software. Even more demanding are devices that directly impact humans, such as medical instruments, robots, and other autonomous systems. They must provide a hierarchical software assurance and the means to prove trustworthiness because there is no room for failure when people could be injured and killed.

For efficiency and effectiveness, there is growing demand to incorporate automation through AI and ML techniques into V&V. While still relevant, traditional validation methods are insufficient to fully test the growing complexity of AI-based systems. Intelligent validation techniques will automate complete testing or certain aspects of it.[1,3,5] Validating software becomes even more complex once AI and ML algorithms are utilized in learning control and technical systems.

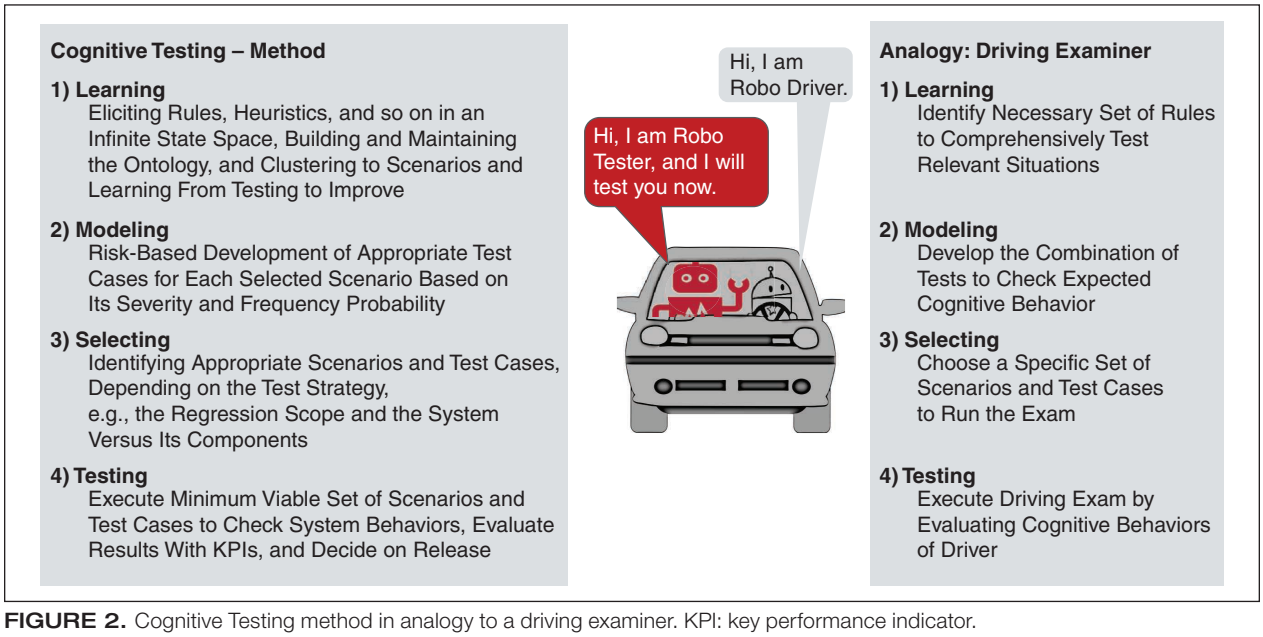

FIGURE 2. Cognitive Testing method in analogy to a driving examiner. KPI: key performance indicator.

ABOUT THE AUTHORS

CHRISTOF EBERT is the managing director of Vector Consulting Services, Stuttgart, 70499, Germany. Contact him at https://twitter.com/christofebert or christof.ebert@vector.com.

DIVITH BAJAJ is with Robo-Test and Vector Consulting Services, Stuttgart, 70499, Germany. Contact him at divith.bajaj@vector.com.

MICHAEL WEYRICH is the director of the Institute for Industrial Automation and Software Engineering, University of Stuttgart, Stuttgart, 70565, Germany. Contact him at michael.weyrich@ias.uni-stuttgart.de.

This validation includes the following types:

- requirements-based testing
- ontology-based testing
- CT.

As an example of emerging AI-based testing let us briefly look to CT, which combines classic functional coverage derived from requirements with heuristics and learning. Figure 2 provides a case study for the CT of an autonomous vehicle. The AI-driven method eliminates potential errors associated with manual derivations since humans may fail to identify and think about certain scenarios. Using heuristics and automatic corner case identification eliminates the enormous amount of time that needs to be invested to derive test cases for brute force validation.[5]

Truly transparent validation methods and processes for testing assume the utmost relevance and will be challenged by technology's step-by-step progress toward autonomous behavior. Although relevant, traditional validation methods are inadequate to completely test the growing complexity of autonomous cars. ML with situational adaptations and software updates and upgrades demands novel regression strategies.

Testing matters—for all of us. It is not a separate activity after development but an integral part, from requirements engineering onward. The relevance of testing and the need for novel test methods is increasingly understood by most developers and responsible companies. This means that we must continuously learn about V&V approaches, that we orchestrate test tools within our application life cycle management/product life cycle management pipelines, and that we position V&V processes in our systems engineering, design, and related activities. Gone are the days of throwing code over the fence to testers. Gone also are the remote V&V teams and their ping-pong with defects and open issues. Testing is no longer a low-priority activity to be outsourced. It is central to software development but with high automation.

Legal exposure, in particular, is driving more orchestrated and systematic testing. Selecting the right test strategy, method, and technology depends on many factors. Every organization needs to tailor its methodology and development environment. While technology is often associated with tools, software organizations must first build the necessary V&V competences. Too often, we see complex tool chains but no tangible test strategy and systematic processes. The following questions must be answered:

- What are your release criteria other than time?
- What is your regression strategy, and how do you determine the necessary regression test cases?
- How do you systematically test for critical corner and feature correlations?
- How do you measure the quality of your software and improve it?

To reduce human risks, software-defined systems must increasingly

be capable of automatically detecting their own defects and failure points. Testing needs experienced practitioners with the right tool chain. A tool is necessary but not sufficient, as we learn from an anecdote about a hunter. One day, when a hunter arrived with his advanced technology, perhaps a bow and arrow or a gun, a tiger cub was very afraid. However, the tigress taught her cub how to sneak and attack. The hunter was never heard from again—which proves that tools are fine but that they will never replace the right process.

References

1. "World Quality Report (WQR) 2021-22," Capgemini, Paris, France, 2021. [Online]. Available: https://www.capgemini.com/research/world-quality-report-wqr-2021-22/
2. D. Graham *et al.*: *Foundations of Software Testing*: *ISTQB Certification*, vol. 4. Andover, U.K.: Cengage Learning, 2019.
3. S. Goericke, *The Future of Software Quality Assurance*. Heidelberg, Germany: SpringerOpen, 2020.
4. *Standard for Software Life Cycle Processes*, ISO 12207-2017, International Organization for Standardization, Geneva, Switzerland, 2017. [Online]. Available: https://www.iso.org
5. *Systems and Software Engineering, Systems and Software Quality Re-quirements and Evaluation (SQuaRE), Quality Requirements Framework*, ISO/IEC 25030-2019, International Organization for Standardization, Geneva, Switzerland, 2019. [Online]. Available: https://www.iso.org
6. *Software Testing*, International Organization for Standardization, Geneva, Switzerland, ISO/IEC/IEEE 29119, 2013.
7. C. Ebert and R. Ray, "Test-driven requirements engineering," *IEEE Softw.*, vol. 38, no. 1, pp. 16–24, Jan. 2021, doi: 10.1109/MS.2020.3029811.
8. C. Ebert and M. Weyrich, "Validation of autonomous systems," *IEEE Softw.*, vol. 36, no. 5, pp. 15–23, Sep. 2019, doi: 10.1109/MS.2019.2921037.

Digital Object Identifier 10.1109/MS.2022.3177874