SILESIAN UNIVERSITY OF TECHNOLOGY PUBLISHING HOUSE

SCIENTIFIC PAPERS OF SILESIAN UNIVERSITY OF TECHNOLOGY 2023
ORGANIZATION AND MANAGEMENT SERIES NO. 186

PRINCIPLES OF SOFTWARE DIAGNOSIS AND TESTING USING TEST AUTOMATION TOOLS

Ireneusz J. JÓŹWIAK[1*], Piotr P. JÓŹWIAK[2], Jan SWITANA[3]

[1] General T. Kościuszko Military University of Land Forces in Wrocław; ireneusz.jozwiak@awl.edu.pl, ORCID: 0000-0002-2160-7077
[2] Wroclaw University of Science and Technology; piotr.jozwiak@pwr.edu.pl, ORCID: 0000-0002-5325-3728
[3] Wroclaw University of Science and Technology, Faculty of Microsystem Electronics and Photonics; switana.jv@gmail.com, ORCID: 0000-0001-7188-8199
* Correspondence author

Purpose: The aim of the article is to provide an in-depth discussion of traditional and modern methods of software diagnostics and testing. The basic ways of implementing these processes will be discussed. We will also discuss the concept of automation of the testing process and its impact on modern software development with usage of Artificial Intelligence.
Design/methodology/approach: The analysis of the article's topic is based on the analysis of sources discussing methods of software diagnostics and testing.
Findings: The importance of analytical thinking and predictive skills.
Originality/value: Presentation of various factors influencing the quality of software production with a particular focus on software test automation.

Keywords: software engineering, testing, diagnostics, automation.

Category of the paper: General review.

1. Introduction

A person responsible for preparing and conducting various tests on applications provided by programmers is referred to as a software tester. This process is mainly based on the technical specification provided by the client. It's important for the tester to identify potential areas of defects based on the specification before software production begins (Cińcio-Pętlicka, 2023). This is a commonly used programming approach known as Test Driven Development (TDD).

There are two main approaches to software testing: manual testing and automated testing. The work of a manual tester involves physically diagnosing the software to verify its functionality. On the other hand, an automated tester designs and implements a system that checks the correctness of the tested software's operation (Praca testera..., 2023). Therefore, high requirements in terms of both hard and soft skills are placed on a software tester.

In recent years, there has been a strong trend towards full test automation, which is not only a result of optimizing software production costs but also becomes a necessary requirement in a world focused on frequent releases of new application versions. The goal is to achieve a situation in which, according to DevOps methodology, every change made by a developer is deployed to production as quickly as possible (Kim et al., 2016). This is, of course, an ideal state but very challenging to attain. It requires having highly specialized teams of programmers and testers capable of implementing full automation of software production processes at every stage. Research indicates that software testing consumes up to 50% of resources, with costs accounting for 50%-60% of all expenses (Myers, 2011; Ramler, 2006). Given the above, achieving full automation of software production processes is not easy and often results in failure or prolonged DevOps implementation processes. According to Statista reports, the adoption of DevOps methodology led to an increase from 33% of companies in 2017 to 63% in 2020 (Statista, 2023).

2. Principles for test development

The International Software Testing Qualifications Board (ISTQB) (2018) has developed and introduced 7 fundamental principles of software testing (Podraza, 2023; Stelmach, 2023; 7 Principles, 2023). The following set establishes basic principles for test development (ISTQB, 2018; ISTQB Syllabus, 2011), even in the case of their automation:

1. Software testing diagnoses defects, not their absence.

Software testing is not intended to prove the absence of defects in the software. The purpose of software testing is to identify the presence of defects. There is a fundamental difference in the approach to testing and test planning. Testing significantly helps reduce the number of undiscovered defects hidden in the software. Finding and resolving these issues is not in itself proof that the software or system is completely free from them (Podraza, 2023).

2. Achieving complete testing of software is impossible.

A crucial aspect of conducting software testing is to be guided by risk analysis and task prioritization. This stems from the fact that time constraints often prevent comprehensive and exhaustive testing of all possible scenarios. Hence, priorities need to be established, strategies devised, and tests well-designed to avoid delaying the system deployment or the intended application.

3. Early initiation of software testing.

Early initiation of software testing increases the likelihood of detecting errors and reduces the cost of their correction. It's best to begin testing during the planning and analysis stage. Finding errors at this point and rectifying them is much less costly than when developers have already started writing code. At that stage, rectifying an error might require revising parts or even the entire application (Stelmach, 2023).

4. Accumulation of software errors.

"The 80% of errors in software are found in 20% of modules." This statement highlights that if an error is detected in a certain part of the software (in a specific module), it's worthwhile to focus on testing the entire module. This is due to the higher logical complexity of specific modules, leading to a domino effect in case of defects. The principle number 3 mentioned earlier is helpful in this context, as such error-prone "hotspots" can be identified during the earliest testing stages.

5. The paradox of continuous software testing updates.

Repeatedly conducting the same tests becomes ineffective. You cannot keep using the exact same tests for the same parts of the software. Tests need to be updated to cover different test cases concerning specific parts of the software or system. This allows for the discovery of new software errors (Syllabus, 2023).

6. Context-dependent software testing.

The approach to testing largely depends on the purpose of the software. Testing is carried out in different ways in different contexts (Syllabus, 2023).

7. Misconception of software being error-free.

It's not enough for software testing to merely pass without errors. It's also crucial to verify whether performance and functionality are appropriate for the client's requirements and capabilities. This means that meeting customer expectations and requirements is just as important as the reliable operation of the program (Podraza, 2023). After all, software that is error-free but too complex to efficiently perform its intended task would be of no use (Stelmach, 2023).

3. The application of process models in an IT project

One of the factors that significantly enhances the effectiveness and ease of work for software testers is the increased efficiency of project teams. This is because the higher the quality of the code, the lower the risk of software errors. Quality, in this context, refers to the accuracy and performance of the program. Therefore, well-organized teamwork among testers, programmers, and architects has a significant impact on the overall software quality as a product. Cockburn (2006) presented a chart that visualizes the path that can be achieved by increasing the efficiency of project teams. The chart outlined in Cockburn's work shows that increased tolerance for uncertainty allows for the implementation and adoption of agile methodologies, incremental development, and iterative methods with greater ease. This leads to improved software performance.

Cockburn (2006) also presents seven principles that are helpful in designing and evaluating software production methodologies:

1. The fastest and cheapest way to communicate is face-to-face interaction.
2. A methodology with too broad a scope incurs costs.
3. The larger the project team, the greater the need for a comprehensive software production methodology.
4. Increased "ceremony" (formality) is only necessary in projects with high criticality.
5. Frequent feedback reduces the need for compromises in software production.
6. Discipline, skills, and understanding are necessary attributes for executing processes, formalizing, and documenting software creation.
7. Efficiency is not the most important factor in software creation activities.

Hence, one might ask why testing, or rather its techniques, works. The answer can be traced back to the fact that in the 1980s, Boehm (Boehm, 1981) demonstrated that the cost of fixing a software error increases exponentially depending on the time of its discovery. For instance, finding a software error within, let's say, 3 minutes of its introduction into the system might not cost us much. However, if the problem is only discovered three months after its introduction, the cost of rectification could be substantial. At that point, not only does the software itself need to be fixed, but also the consequences of the software functioning with the error need to be addressed.

4. Automation of tests depending on their application method

Over the years, various software testing methods have been developed. In this chapter, they will be described in the context of automation capabilities.

The automation of tests depending on their application method is a strategy that aims to optimize the testing process by selectively automating certain types of tests based on their characteristics and requirements. This approach acknowledges that not all tests are equally suitable for automation and that the decision to automate should be made considering the specific context and goals of the testing effort.

Tests can vary widely in terms of their scope, complexity, and objectives. Some tests, such as unit tests, focus on isolating and verifying small units of code in isolation, while others, like end-to-end tests, aim to validate the entire software system's functionality.

The following testing techniques differ in scope and the approach in which they are applied. As a result, by utilizing the testing techniques listed below, various types of software errors can be detected.

4.1. Ad-hoc (exploratory) testing

Ad-hoc tests, also known as exploratory tests, serve both as a means of learning about the software and verifying its functionality. This leads to the proper design of further tests of this type. Such ad-hoc tests are most useful in projects with limited documentation (What is Exploratory Testing, 2023). Unfortunately, due to their nature, this type of testing is not suitable for automation. Automation requires complete knowledge of the subject under test. Each test acts as a contract describing how a specific part of the software should function. If we don't have prior knowledge of the system's behavior, exploratory tests can be used as a preliminary step in creating documentation.

4.2. Unit tests

Unit testing involves testing individual modules, their functions, to verify their proper operation for specific input data. A key characteristic of this type of testing is breaking down the program into distinct unit tests (Unit test, 2023). It's the most common type of testing generated within automation environments like Continuous Integration and Continuous Delivery (CI/CD). It holds fundamental importance as it often develops in parallel with implemented functionalities. Unit tests focus on testing small sections of code and serve as a valuable source of software documentation, succinctly and systematically describing the functionality of individual parts within the whole system. The significance of these tests is reflected even in integrated development environments like IntelliJ or Visual Studio Code, which integrate their user interfaces to facilitate the direct execution of unit tests from the programming environment, making it easier for developers to run tests multiple times.

In practice, unit tests are written in the same programming language as the specific part of the application. Unit tests utilize a set of pre-existing libraries that aid in the testing process, allowing programmers to write concise testing procedures that examine small portions, often a single procedure or function in the code. An issue in unit testing is isolating a small code fragment from its connections to the rest of the system. An example of this is accessing a database. However, libraries that support unit testing often provide the necessary functionalities for simulating other systems, referred to as Mocks.

Unit tests are generally executed in two scenarios. The first is the manual execution of unit tests by the programmer, which checks whether introduced implementation changes have adversely affected the entire system. Only after the programmer successfully passes local verification can they submit their changes to the central code repository. In software development teams that employ agile methodologies or DevOps (Kim et al., 2016) for work organization, the moment code changes are uploaded to the central repository automatically triggers what's called a Pipeline. In the initial step, this Pipeline involves compiling the code with the latest changes, followed by executing a suite of tests. The most commonly used tool for supporting the CI/CD process is Jenkins. However, suitable support for building CI/CD automation can essentially be found in any major code repository tool with automation mechanisms. Examples of such tools include GitHub Actions, GitLab, and AWS CodeBuild.

4.3. Integration tests

Integration tests aim to verify the correctness of the interaction between different modules of the tested software, particularly the interactions of various interfaces that are often closely linked through data exchange. In large systems, there is a significant number of connections between system components, which poses a challenge in implementing integration tests.

There are several types of integration tests that are recognized (Kitakabe, 2023):

- **Big-bang integration testing**: This integration testing involves integrating all the components at once and testing them as a complete system. The method is typically used when the components are relatively independent and can be tested individually.
- **Top-down integration testing**: You can use top-down integration testing when the components are integrated and tested from the highest level to the lowest level. The approach is used when the higher-level components depend on the lower-level components.
- **Bottom-up** integration testing: The integration testing type involves integrating and testing the components from the lowest level to the highest level.
- **Sandwich/hybrid integration testing**: This integration testing involves combining elements of both top-down and bottom-up integration testing. The components are tested from both the top and bottom levels, with stubs and drivers used to simulate the missing components.
- **Continuous integration testing**: It involves continuously integrating and testing the components as they are developed. The method helps to catch and resolve problems early in the development process, improving the overall quality of the system.

The selection of the appropriate type of integration test is linked to the development process and the specific requirements of the system itself. Unfortunately, not all types of integration tests are easily automatable. While automated testing of a system's APIs can be implemented using contract tests, such as Pact, testing the user interface is more challenging to execute. However, this does not mean it's impossible, and one can utilize tools like Selenium for UI testing. Integration tests, in general, require a significant amount of human and machine resources for their implementation and execution.

4.4. Regression tests

Regression testing involves testing after changes, such as bug fixes, software modifications, or the addition of new features. Its purpose is to ensure that the changes introduced to the software have not introduced new errors. These tests also significantly increase the likelihood of identifying issues that were not visible before the changes were made. It's important to emphasize that such tests must be conducted in the same environment and scope as the tests mentioned earlier (Wydmański, 2023).

Automating regression tests, similar to integration tests, is a complex task. Regression tests often appear in later stages of implementation or may even be absent from the technological stack of a given team. As mentioned in the work by Sutapa et al. (2020), some of the most commonly used tools for assisting with the automation of regression tests include Selenium, SAHI, and Robot Framework. However, implementing effective automated regression tests requires careful planning, design, and integration into the development process to ensure that code changes do not inadvertently introduce new issues into the software.

4.5. Performance tests

Performance testing of software is typically conducted as the last phase of testing, following the resolution of defects detected in previous testing stages. Its purpose is to examine the behavior of an application or system under various load conditions. There are different types of performance tests that can be applied to projects with different characteristics. The following types of performance tests are distinguished: load tests, stress tests, scalability tests, spike tests, endurance tests, concurrency tests, and throughput tests (Performance Testing, 2023). The challenge in implementing performance tests lies primarily in selecting the areas that need to be tested. Performance testing can encompass the overall performance of the software, as well as specific aspects like network throughput or data flow. The nature of a specific application guides the choice of the most suitable type of performance tests. Performance testing may involve subjecting the server, database, or the application itself to various types of loads and conditions to evaluate its performance.

To conduct performance testing of software, commonly used tools include JMeter, LoadUI, Gatling, Fiddler, and LoadRunner. These tools assist in simulating various load conditions and scenarios to evaluate the performance of the software.

From the mentioned types of software testing, a conclusion can be drawn that implementing tests for most software types is possible but still highly complex. A natural approach to minimizing the risk of producing high-quality software is to ensure that testing is introduced as early as possible in the software development process. This helps keep technical debt at a low level. Unfortunately, the automation of software testing only seemingly reduces the workload for programmers and testers. In the case of a highly developed system undergoing intensive work, changes in the code can trigger a cascade of adjustments needed to align tests with new realities. Automated software testing requires continuous monitoring and maintenance. This presents a kind of paradox, as automation is intended to reduce manual work. In certain cases, maintaining the automation itself becomes more costly than the initial assumptions. Therefore, a fundamental principle in writing software tests is to ensure their quality and adequacy, so that the number of tests doesn't increase uncontrollably without a corresponding increase in the quality of the developed software.

5. Artificial Intelligence in test automation

Currently, we are witnessing a strong interest in utilizing artificial intelligence (AI) to solve problems that are challenging to address using traditional approaches. In the context of test automation, efforts are also underway to harness artificial intelligence (AI) to develop testing methods that require minimal human involvement in creating and maintaining tests. The progress in utilizing AI in testing has been outlined in the work by Jenny Li and colleagues (Jenny Li, et al., 2020).

In general, AI is most commonly used for test automation in the form of:

- **Self-healing tests**: Traditional test automation tools have specific identifiers to define the components of an application such as locator, usually name, id, Xpath, and type, to run test steps successfully (Shabarish, 2023). When the application is updated or changed in any way, these components can also change. Due to the fixed definitions are given to these elements, the tests functioning fine before the change will now fail and provide a false negative result. Self-healing tests apply artificial intelligence algorithms to automatically identify unexpected errors due to dynamic properties and recommend a better alternative or automatically update the script. This testing stops tests from failing and saves the time a QA might have spent trying to find and fix the issue (Pandey, 2023).
- **Visual Locators**: Usually it is needed to write selectors to target the specific things while interact with during tests. Lots of them. Selectors are critical to test exactly what we are aiming for, but there are challenges. Not only we usually have to write quite a lot of them, but in some cases an obvious selector doesn’t even exist and it is necessary to rig up a creative workaround. It is possible to do it, but that kind of workaround is typically fragile and can easily break as the application develops (Shain, 2021). In AI-based user interface testing, visual locators can now find elements, even when their locators have been altered, on a web application by vision. This eliminates the need for hard coding with Accessibility IDs or other locators. Furthermore, intelligent automation tools can now use OCR and other image recognition techniques to map the application, locate visual regressions, or verify elements (Pandey, 2023).

– **AI Analytics of Test Automation Data**: AI drives automation, performs faster to identify errors and causes, suggests fixes and connect a set of related tests. This not only makes test automation faster but also more precise. AI is capable of automatically accessing data, running tests and identifying errors and other relevant affected tests (Chandrasekharan, 2023). The reason for such a state of affairs is generating by tests copious amounts of data that must be combed to derive meaning. The application of AI to this process dramatically increases its efficiency. More sophisticated applications of AI algorithms can also identify false negatives and true positives in test cases. This can be very helpful and significantly reduce the workload of QAs (Pandey, 2023).

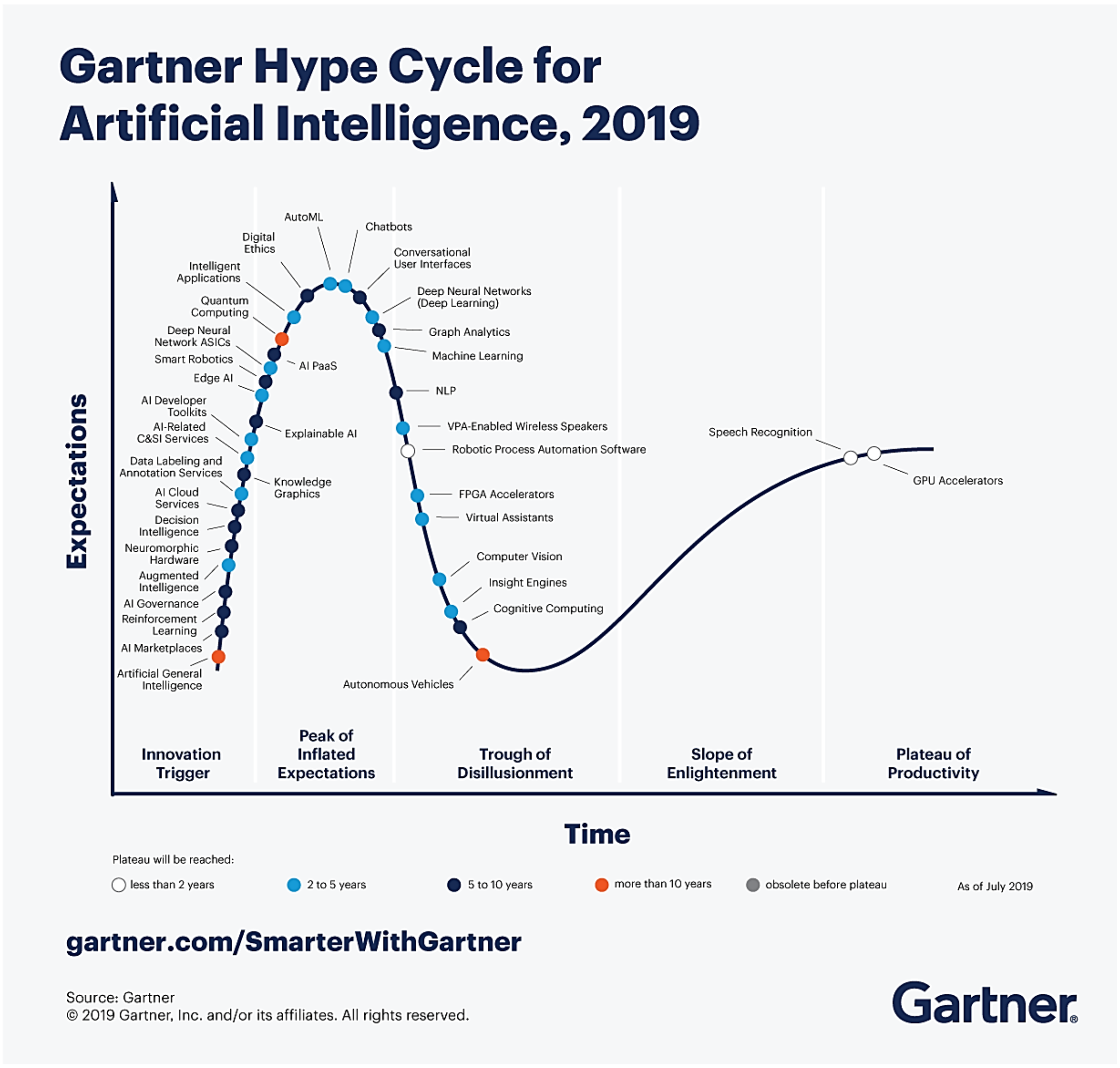

Figure 1. Gartner Hype Cycle for Artificial Intelligence in 2019.

Source: Gartner (2019).

AI-powered test automation is particularly beneficial in complex software projects where manual testing or traditional automation might fall short. However, it's important to note that while AI brings significant advantages, it's not a replacement for traditional testing practices but rather a tool that enhances and complements them. Careful consideration of the application context, data quality, and ongoing training of AI models are essential for successful implementation.

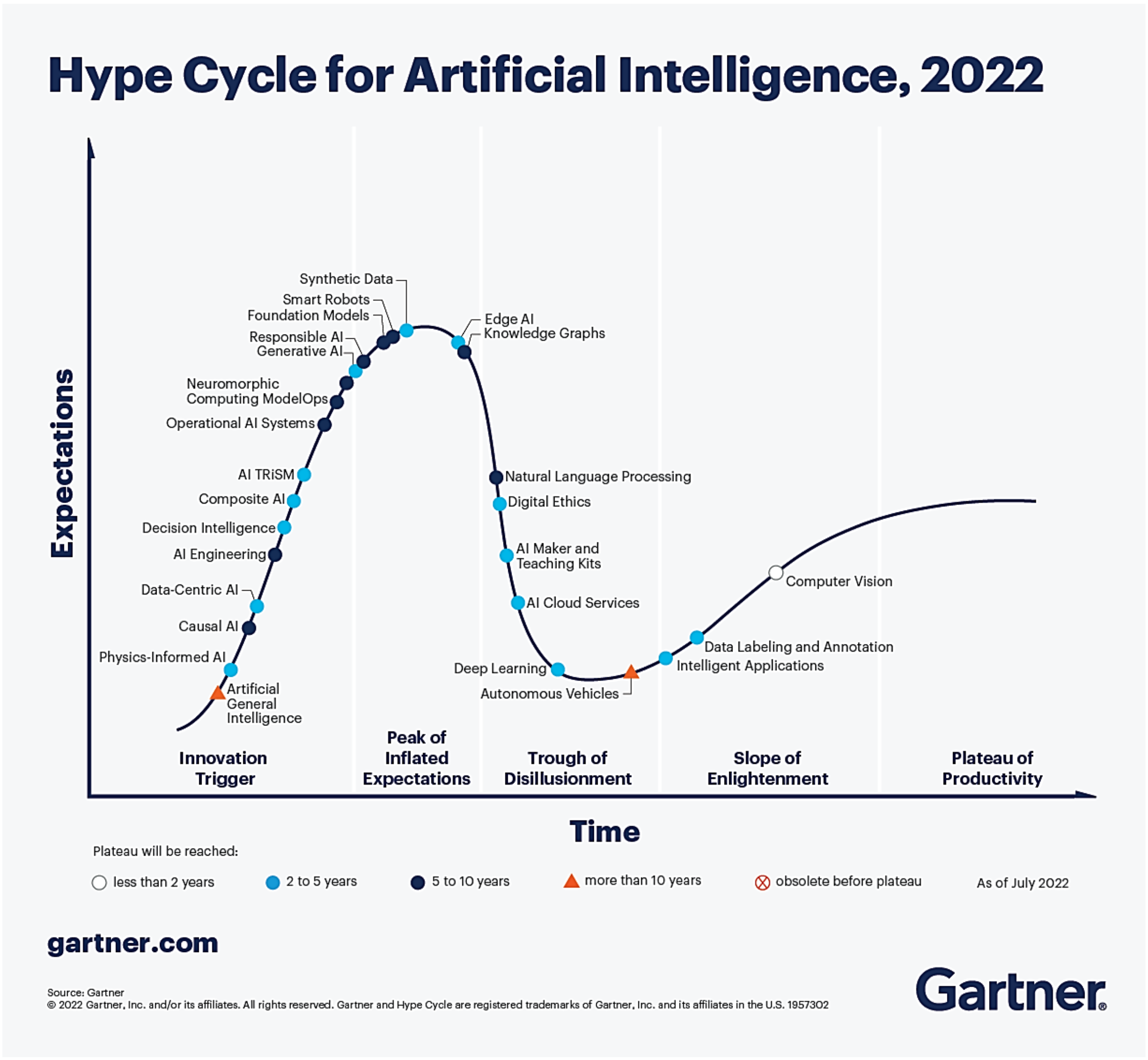

Figure 2. Gartner Hype Cycle for Artificial Intelligence in 2022.

Source: Gartner (2022).

Whether artificial intelligence (AI) will truly be able to assist in the automatic generation and updating of tests is currently difficult to determine, as we are in a peak moment of interest in AI. This is evident from the Gartner's Hype Cycle graphs for year 2019 shown in Figure 1 and in Figure 2 for year 2022. A specific sub-field, deep learning, has caused a lot of this excitement. The graph in Figure 1 indicates that we are in a period of high expectations regarding the potential use of AI. Year 2022 shows similar trends, most AI-related technologies are ahead of peak expectations. However, over time, expectations will likely become more realistic, and the AI technology itself will probably improve the situation related to test automation. It is noteworthy that in the two Gartner diagrams presented, there is no separate category for test automation using AI. The topic is placed in 2019 in the field of AI Developer

Toolkits, while in 2022 it is further generalized to AI Engineering. Both areas are still in the very early stages of development, estimated at five to ten years to mature. However, as always, it won't turn out to be a one-size-fits-all solution for all the challenges associated with it.

References

1. *7 zasad testowania w 5 minut*. Retrieved from: https://projectquality.it/, 20.07.2023.
2. Boehm, B. (1981). *Software Engineering Economics*. Englewood Cliffs, New Jersey: Prentice-Hall.
3. Chandrasekharan, L. (2023). *Scope of AI in Automation Testing: How AI Plays an Important Role?* Retrieved from: https://testsigma.com/blog/scope-and-impact-of-ai-in-automation-testing/, 28.08.2023.
4. Cińcio-Pętlicka, A. (2023). *Czym zajmuje się i ile zarabia tester oprogramowania*. Retrieved from: https://nofluffjobs.com/pl/log/, 18.07.2023.
5. Cockburn, A. (2006). *Agile Software Development. The cooperative game*. Pearson.
6. *Czym jest testowanie eksploracyjne?* Retrieved from: https://testerzy.pl/baza-wiedzy/, 1.08.2023.
7. Gartner (2019). *Top Trends of Gartner Hype Cycle for AI*. Retrieved from: https://www.gartner.com/smarterwithgartner/top-trends-on-the-gartner-hype-cycle-for-artificial-intelligence-2019, 3.08.2023.
8. Gartner (2022). *What's New in Artificial Intelligence from the 2022 Gartner Hype Cycle*. Retrieved from: https://www.gartner.com/en/articles/what-s-new-in-artificial-intelligence-from-the-2022-gartner-hype-cycle, 3.08.2023.
9. ISTQB (2018). *7 zasad testowania*. Retrieved from: https://testerzy.pl/baza-wiedzy/, 8.07.2019.
10. Jenny, L.J., Ulrich, A., Bai, X., Bertolino, A. (2020). Advances in test automation for software with special focus on artificial intelligence and machine learning. *Software Qual. Journal, vol. 28,* pp. 245-248, doi: 10.1007/s11219-019-09472-3.
11. Kim, G., Humble, J., Debois, P., Willis, J. (2016). *The DevOps Handbook*. Portland: IT Revolution.
12. Kitakabe (2023). *Integration testing, a detailed guide*. Retrieved from: https://www.browserstack.com/guide/integration-testing, 3.08.2023.
13. Myers, G.J., Sandler, C., Badgett, T. (2011). *The art of software testing*. New Jersey: John Wiley & Sons.
14. Pandey, S. (2023). *AI Automation and Testing*. Retrieved from: https://www.browserstack.com/guide/artificial-intelligence-in-test-automation, 10.08.2023.

15. Podraza, R. (2023). *Siedem zasad testowania. Co oznaczają?* Retrieved from: https://testowanie-oprogramowania.pl/, 20.07.2023.
16. *Praca testera oprogramowania, gdzie jej szukać i jakie zarobki może zaoferować*. Retrieved from: https://zwierciadlo.pl/material-partnera, 18.07.2023.
17. Ramler, R., Wolfmaier, K. (2006). *Economic perspectives in test automation: balancing automated and manual testing with opportunity cost*. Proceedings of the International Workshop on Automation of Software Test, pp. 85-91.
18. Shabarish (2023). *Self Healing Test Automation*. Retrieved from: https://testsigma.com/blog/self-healing-tests-maintenance-testsigma/, 28.08.2023.
19. Shain, D. (2021). *Why You Should Use Visual AI Locators Instead of Fragile Selectors in Your Tests.* Retrieved from: https://applitools.com/blog/why-visual-locators-not-selectors-in-tests/, 28.08.2023.
20. Statista (2023). Retrieved from: https://www.statista.com/topics/9369/devops/#topic Overview, 10.08.2023.
21. Stelmach, T. (2023). *Siedem zasad testowania oprogramowania*. Retrieved from: https://qualityisland.pl/, 20.07.2023.
22. Sutapa, F.A., Kusumawardani, S.S., Permanasari, A.E., (2020). A Review of Automated Testing Approach for Software Regression Testing. *IOP Conf. Ser.: Mater. Sci. Eng., 846*, 012042, doi:10.1088/1757-899X/846/1/012042.
23. Sylabus ISTQB – Poziom podstawowy (wersja 2011.1.3) (2011). Retrieved from: http://getistqb.com/docs/, 20.07.2023.
24. *Testowanie wydajności. Teoria*. Retrieved from: https://testerzy.pl/baza-wiedzy/artykuly/, 1.08.2023.
25. *Unit test basics*. Retrieved from: https://learn.microsoft.com/en-us/visualstudio, 31.07.2023.
26. Wydmański, M. (2023). *Testy potwierdzające i regresyjne – jak przebiegają w praktyce.* Retrieved from: https://craftware.pl/testy-potwierdzajace-i-regresyjne/, 1.08.2023.