https://doi.org/10.31891/2219-9365-2026-87-35
UDC 004.415.5:004.8

PECHENENKO Vladyslav
National University «Kyiv Aviation Institute»
https://orcid.org/0009-0002-2447-6831
e-mail: v.o.pechenenko@gmail.com
GRINENKO Olena
National University «Kyiv Aviation Institute»
https://orcid.org/0000-0001-9673-6626
e-mail: gsa_ck@ukr.net

# TEST-DRIVEN DEVELOPMENT ENHANCED BY ARTIFICIAL INTELLIGENCE: AN APPROACH TO SMARTER TEST DESIGN AND VALIDATION

*In order to enhance the caliber, applicability, and maintainability of automated tests, the study investigates the incorporation of Artificial Intelligence (AI) approaches into the Test-Driven Development (TDD) methodology. The time needed for manual test design, the challenge of finding non-trivial edge situations, and the increasing maintenance cost in large codebases sometimes limit the practical adoption of TDD, despite the fact that it is still one of the most well-known approaches in agile software engineering. Therefore, the paper suggests an AI-enhanced TDD strategy that uses machine learning, massive language models, and natural language processing to enable requirement interpretation, test development, defect-prone region discovery, and test-suite optimization. The suggested method, in contrast to fully autonomous testing scenarios, maintains the developer's primary role: AI is viewed as an intelligent assistant that recommends candidate tests, ranks risky scenarios, clarifies coverage gaps, and supports refactoring decisions, while human control over final validation is maintained. The article outlines a workable workflow for implementing the method on an ASP.NET Core password-reset module, specifies the best methods for each stage, and systematizes how AI may be included into the traditional Red-Green-Refactor cycle. Benefits, restrictions, and adoption concerns are discussed along with an organized mapping between TDD stages and AI capabilities. Faster initial test generation, more coverage of border and negative scenarios, improved verification effort prioritization, and reduced long-term maintenance costs are the primary anticipated outcomes of the suggested technique. The research contends that data quality, controllability of generated outputs, and rigorous developer review are critical to the efficacy of AI-enhanced TDD. Researchers and practitioners interested in AI-assisted testing, intelligent software engineering, and modernizing quality-assurance procedures in software projects may find value in the produced results.*

*Keywords: software testing, software quality, defect prediction, automated test generation, artificial intelligence, test-driven development.*

ПЕЧЕНЕНКО Владислав, ГРІНЕНКО Олена
Національний університет «Київський авіаційний інститут»

# РОЗРОБЛЕННЯ, КЕРОВАНЕ ТЕСТУВАННЯМ, ПОСИЛЕНЕ ШТУЧНИМ ІНТЕЛЕКТОМ: ПІДХІД ДО ІНТЕЛЕКТУАЛЬНОГО ПРОЄКТУВАННЯ ТА ВАЛІДАЦІЇ ТЕСТІВ

*У дослідженні розглянуто інтеграцію підходів штучного інтелекту (ШІ) у методологію розроблення, керованого тестуванням (Test-Driven Development, TDD), з метою підвищення якості, релевантності та супроводжуваності автоматизованих тестів. Попри те, що TDD залишається одним із найпоширеніших підходів гнучкої інженерії програмного забезпечення, його практичне застосування може обмежуватися значними витратами часу на ручне проєктування тестів, складністю виявлення нетривіальних граничних випадків і зростанням вартості супроводу тестів у великих програмних кодових базах.*

*Запропоновано стратегію TDD, посилену штучним інтелектом, яка поєднує методи машинного навчання, великі мовні моделі та оброблення природної мови для інтерпретації вимог, формування тестів, виявлення ділянок коду, схильних до дефектів, і оптимізації тестових наборів. На відміну від повністю автономних сценаріїв тестування, запропонований підхід зберігає провідну роль розробника. ШІ розглядається як інтелектуальний помічник, здатний пропонувати варіанти тестів, ранжувати ризикові сценарії, виявляти прогалини в тестовому покритті та підтримувати ухвалення рішень щодо рефакторингу, тоді як остаточна валідація залишається під контролем фахівця.*

*У статті подано практичний робочий процес реалізації підходу на прикладі модуля відновлення пароля ASP.NET Core, визначено доцільні методи для кожного етапу та систематизовано способи інтеграції ШІ у традиційний цикл Red–Green–Refactor. Розглянуто переваги, обмеження та організаційні аспекти впровадження, а також сформовано відповідність між етапами TDD і можливостями штучного інтелекту. Очікуваними результатами застосування запропонованого підходу є пришвидшення первинного генерування тестів, ширше охоплення граничних і негативних сценаріїв, ефективніше визначення пріоритетів верифікації та зниження довгострокових витрат на супровід програмного забезпечення.*

*Обґрунтовано, що результативність TDD, посиленого штучним інтелектом, залежить від якості вхідних даних, контрольованості згенерованих результатів і ретельної експертної перевірки з боку розробника. Отримані результати можуть бути корисними для дослідників і практиків у сфері тестування програмного забезпечення з підтримкою ШІ, інтелектуальної інженерії програмного забезпечення та модернізації процедур забезпечення якості програмних продуктів.*

*Ключові слова: тестування програмного забезпечення, якість програмного забезпечення, прогнозування дефектів, автоматизоване генерування тестів, штучний інтелект, розроблення, кероване тестуванням.*

Стаття надійшла до редакції / Received 10.06.2026
Прийнята до друку / Accepted 22.07.2026
Опубліковано / Published 10.09.2026



## INTRODUCTION

It is anticipated that modern software products will advance rapidly while retaining a high degree of dependability. By requiring tests to be created before functional code, Test-Driven Development (TDD) helps agile teams accomplish this aim by making expected behavior explicit and lowering the likelihood of undetected regressions. However, when requirements are ambiguous, there are many edge situations, and developers have to maintain a continuously expanding set of tests alongside production code, the disciplined application of TDD becomes challenging in real projects [1], [2].

These challenges are particularly evident in distributed systems, enterprise applications, and integration-heavy solutions, where test design must take into account non-functional requirements, concurrency, invalid inputs, external service failures, and temporal constraints in addition to straightforward assertions. One of the primary benefits of TDD is undermined as a result of developers having to either devote a significant amount of time to creating and maintaining tests or cutting back on testing thoroughness.

Recent developments in AI offer fresh ways to overcome these constraints. Natural language processing can formalize user stories and acceptance criteria; machine learning can identify code areas that are more likely to contain defects; and large language models can generate candidate unit tests, fixtures, and assertions from code and documentation. The combination of these techniques creates the basis for a more adaptive and knowledge-aware form of TDD that preserves developer control while increasing productivity and quality.

## LITERATURE REVIEW

The methodological foundations of TDD were formulated by K. Beck, who described the Red-Green-Refactor cycle as a disciplined way to connect design, implementation, and automated verification [1]. Later studies showed that TDD can improve code quality and developer confidence, but also highlighted costs related to learning, discipline, and maintenance overhead [2].

Reducing the manual burden of testing has been the focus of several research initiatives. Systematic path exploration can identify flaws that are challenging to find manually, as shown by symbolic execution and associated automated techniques [3]. Model-based testing offered ways to derive tests from codified system behavior [4], while search-based software testing demonstrated the efficacy of evolutionary and optimization-based approaches for creating or improving tests under difficult constraints [5].

Defect prediction is another crucial area. Machine learning models can determine which components are more likely to fail and hence require more testing attention by examining code metrics, change history, and repository activity [6]. Simultaneously, studies on natural language-driven test generation have demonstrated that when appropriate intermediate representations are employed, software requirements specified in natural language may be converted into test objects [7].

Interest in AI-assisted test generation has increased with the development of big language models. LLMs can provide compilable, somewhat diversified unit tests with considerable coverage, according to recent empirical investigations [8], [9]. However, because generated assertions may be weak, redundant, or not adequately aligned with domain constraints, they still need human inspection. However, mutation testing is still a crucial method for determining whether a test suite truly finds injection errors instead of just running code paths [10].

Therefore, previous research both supports the importance of automation in software testing and indicates that no single method is enough on its own. This necessitates an integrated methodology that integrates test-suite quality management, automated test generation, risk-oriented prioritization, and requirement comprehension inside the TDD workflow.

This article aims to develop and elucidate a TDD methodology that integrates AI to consolidate defect prediction, automated test generation, requirement analysis, and test optimization into a cohesive process for contemporary software projects.

## METHODOLOGY

The suggested method incorporates AI support into every stage of Red-Green-Refactor, extending the traditional TDD process. AI is used to enhance analytical depth and decrease the amount of regular work involved in planning, prioritizing, and updating tests rather than to replace the developer.

Functional requirements, user stories, and acceptance criteria are converted into organized, testable representations during the requirement-understanding phase. Entities, actions, constraints, preconditions, and anticipated results are all detected via natural language processing. Practically speaking, this implies that a narrative like "a user can reset a password by e-mail within 15 minutes" is broken down into clear business rules, validation restrictions, and other routes that can subsequently be converted into automated tests.

Large language models use specified requirements, public interfaces, source code, and available examples to generate candidate unit and integration tests throughout the test-generation stage. Test names, fixtures, mocks, input data, assertions, and negative scenarios are examples of the created artifacts. Nevertheless, the output is not automatically approved. The developer examines the suggested tests, eliminates unnecessary cases, fortifies claims, and modifies test structure and naming to conform to project standards.

Machine learning is utilized during the risk-analysis stage to determine what needs to be tested first. Signals suggesting a module may be more prone to errors include code churn, dependency structure, complexity indications, recent changes, and historical flaws. This enables the team to concentrate early validation efforts on code segments where it is most expensive to overlook a flaw. In this sense, AI helps with both the strategic allocation of testing work and the number of tests developed.

The resulting suite is assessed and improved during the optimization step using search-based techniques and mutation testing. The corresponding test can be reinforced or broadened if a mutant survives. Search-based strategies may propose more diverse inputs, boundary values, and parameter combinations than those initially suggested by the LLM. Consequently, the test suite grows in size and becomes more sensitive to errors.

The general workflow of the proposed approach is presented in Table 1.

Table 1

**Integration of AI techniques into the TDD cycle**

| TDD stage | Primary input | AI support | Expected effect |
|---|---|---|---|
| Red | User stories, requirements, interfaces | NLP extracts constraints; LLM proposes candidate tests, mocks, and edge cases | Faster creation of the initial test set and clearer behavior specification |
| Green | Failing tests and implementation feedback | Risk model highlights defect-prone modules and missing scenarios | More targeted implementation and earlier validation of risky code |
| Refactor | Passing tests, code changes, and historical data | Mutation testing, search-based input generation, and AI review of brittle or redundant tests | Stronger assertions, reduced redundancy, and improved maintainability |
| Continuous feedback | Execution history, coverage, mutation score | AI summarizes gaps, explains anomalies, and recommends updates after code evolution | Lower maintenance effort and better long-term relevance of the suite |

To demonstrate the practical applicability of the proposed methodology, the article considers an ASP.NET Core password-reset module as an illustrative case.

Examine an ASP.NET Core password-reset module to illustrate the practical reasoning behind the suggested method. The functional requirements include a valid e-mail address, token expiration after 15 minutes, single-use of a token, and request rate limiting. In a traditional TDD workflow, the developer manually derives tests for valid and invalid e-mail input, expired tokens, repeated token use, and throttling behavior. The same workflow is used in the suggested method, starting with the automatic creation of test skeletons for typical and boundary instances and the AI-assisted extraction of these rules from the textual requirement.

Tests for successfully submitting reset requests, rejecting faulty addresses, invalidating expired tokens, and guarding against repetitive requests from the same account, for instance, might be included in the created suite. Next, the developer verifies that these tests accurately capture the application's actual architecture, domain invariants, naming conventions, and error semantics. In this case, the developer guarantees semantic accuracy and project-specific quality while AI reduces the time required to create a comprehensive first suite.

The function of risk-based prioritization is likewise demonstrated by the same situation. The relevant tests can be run early and examined more thoroughly if repository history reveals frequent changes in token handling or external email integration. Because not every module is equally dangerous, this makes the method especially helpful in iterative development.

In practical applications, xUnit or NUnit may be employed to execute tests, Moq or NSubstitute may be utilized to isolate dependencies, Coverlet serves to assess structural coverage, and Stryker.NET is applicable for conducting mutation analysis. This toolchain is advantageous as it distinguishes the creation of candidate tests from the validation of their efficacy. The AI layer can suggest tests and explain what they are meant to do, but the testing toolchain should provide clear quality indicators that decide whether those tests are deterministic, maintainable, and sensitive enough to faults.

## RESULTS

The minimization of repetitious cognitive work during test preparation is the main benefit of AI-enhanced TDD. Under time constraints, AI can swiftly list similar classes, erroneous inputs, and probable boundary conditions

that could otherwise be overlooked. Additionally, it facilitates the connection between requirements and implementation artifacts, enhancing automated verification and traceability between intended behavior.

Adaptability is an additional benefit. AI models are appropriate for teams that deal with shifting requirements, polyglot codebases, and numerous testing levels since they can assess both plain language and source code. In these situations, the suggested method facilitates the creation of integration and API-oriented checks in addition to unit testing.

However, there are a number of restrictions that need to be taken into account. Defect-prediction models heavily rely on the quality and representativeness of project history; LLMs may overproduce "happy-path" scenarios; and generated tests may be syntactically valid but semantically weak. Organizations must also take into account the risk of relying too much on opaque model outputs, confidentiality, timely governance, and reproducibility of outcomes. Therefore, rather than replacing engineering judgment on its own, AI-assisted testing should be implemented as an augmentation technique.

## DISCUSSION

Instead of replacing the entire process, AI-enhanced TDD should be implemented gradually for practical acceptance. Using AI to create candidate unit tests for modules with reliable interfaces and clear business rules is a sensible starting point. This makes it possible for a team to assess the value of created tests, the caliber of assertions, and the extra review effort without having to alter the entire quality-assurance procedure all at once.

Establishing clear review criteria for tests created by AI is the second suggestion. In terms of name, isolation, determinism, assertion quality, handling of edge cases, and readability, teams should specify what makes a test acceptable. Because AI systems can generate outputs that are nominally valid but weak or noisy, such criteria are required. The risk of building up low-value tests is greatly decreased by a straightforward approach requiring each generated test to be examined, run, and assessed against coverage or mutation indicators.

Project data is the subject of the third suggestion. Only when repository history, issue tracking data, and code metrics are sufficiently clean and consistent can defect prediction and risk-oriented prioritization be applied. Risk scores should be regarded as advising rather than authoritative if such data are lacking. Organizations should also decide which items can be transferred to external AI systems and which, because of confidentiality concerns, must stay inside protected infrastructure.

Lastly, striking a balance between automation and accountability is essential for successful deployment. AI can speed up the identification of potential scenarios and cut down on tedious work, but the development team must still be in charge of the behavior that the test suite verifies. As a result, the most advanced type of AI-enhanced TDD is a regulated collaboration model where the developer, not the tool, makes all final decisions during the testing process.

The anticipated outcomes of the suggested methodology can be evaluated using a composite of productivity, quality, and maintainability metrics instead of relying on a singular measure. An effective assessment of AI-enhanced Test-Driven Development should integrate measures of productivity, quality, and maintainability instead of depending on a solitary statistic. The initial set of indications include the duration necessary to generate the initial test suite, the proportion of tests approved by the developer without significant modification, and the work required to sustain tests following alterations in the production code. These indicators indicate if AI genuinely diminishes repetitious tasks rather than simply reallocating effort from test creation to test evaluation.

The second category of indications pertains to the quality of the generated tests. In this context, statement coverage, branch coverage, mutation score, defect leakage, and the ratio of edge-case scenarios are particularly significant. Coverage alone cannot demonstrate the robustness of the tests; hence, mutation analysis should be employed to ascertain whether the resulting assertions can identify deliberately introduced flaws. A superior mutation score, coupled with consistent execution and comprehensible tests, indicates greater practical utility than mere surface increases in coverage.

The third category of indicators pertains to decision support. Defect-proneness models assess modules based on historical risk, allowing the quality of prioritizing to be validated by contrasting anticipated high-risk modules with the actual defect distribution found in subsequent iterations. From a pragmatic perspective, even modest prediction accuracy can be advantageous if it assists teams in determining where to allocate additional review resources, where to produce more negative tests, and which components of the system necessitate early regression assessments in the CI pipeline.

In the ASP.NET Core password-reset example presented in this article, the most comprehensive evaluation package would encompass the quantity of generated candidate tests, the proportion of accepted tests post-review, execution stability, mutation score for token-validation logic, and the number of defects mitigated prior to integration. This package enables the evaluation of the suggested strategy both conceptually and in terms of engineering feasibility and operational value.

The interpretation of these anticipated results necessitates caution, as the practical utility of AI-enhanced TDD is contingent upon contextual and organizational variables.

Multiple challenges to validity must be considered when evaluating the anticipated advantages of AI-enhanced Test-Driven Development (TDD). The quality of AI output is significantly influenced by the thoroughness of requirements, the quality of naming in source code, and the accessibility of representative historical data. If user stories lack clarity or the repository history is cluttered, the ensuing tests and risk assessments may be unreliable. Under such circumstances, AI may generate seemingly credible yet ineffective tests that foster a misleading sense of assurance.

Secondly, findings derived from a specific technology stack cannot be universally applied to all software systems. Web APIs with clear contracts are more appropriate for AI-assisted test generation than systems characterized by significant hidden states, legacy dependencies, or intricate hardware interactions. Consequently, the suggested methodology ought to be regarded as a versatile framework, with its specific application contingent upon project design, testing culture, and availability of high-quality development data.

Third, organizational and ethical limitations may restrict immediate implementation. Repository contents, issue trackers, and production logs frequently contain sensitive business information; thus, teams must establish explicit guidelines for timely formulation, data anonymization, and the utilization of external AI services. In industrial settings, governance issues are as critical as technical ones; the advantages of acceleration must not come at the expense of information leakage or less accountability for quality judgments.

A pragmatic prerequisite for a successful implementation is the presence of a review policy that explicitly delineates the criteria for the acceptance, revision, or rejection of AI-generated tests. This protocol may encompass criteria for determinism, assertion robustness, name precision, fixture integrity, and the inclusion of negative scenario coverage. When these requirements are well defined, AI is more seamlessly integrated into a team's everyday workflow and is less prone to producing fragile artifacts that elevate maintenance burdens. These factors also present other intriguing avenues for more research and practical implementation.

Subsequent research should transition from theoretical models to systematic empirical comparisons of manual TDD and AI-augmented TDD on representative software tasks. These studies may evaluate time-to-first-test, enhancement of mutation scores, quantity of escaped flaws, and the maintainability of generated tests across various technological stacks. Benchmark-based assessment would facilitate the identification of areas where AI delivers substantial engineering value and where its impact is confined to draft creation.

A viable avenue is the incorporation of the proposed methodology into educational practice. In programming classes, AI-assisted Test-Driven Development (TDD) can facilitate the rapid generation of tests while elucidating the necessity of particular assertions, fixtures, or negative scenarios. In this capacity, AI functions as an interactive mentor that facilitates thinking on software quality, rather than merely serving as a code generation tool. This can enhance the linkage among requirements analysis, implementation, and verification for students.

From an industrial perspective, the subsequent stage involves integrating AI-assisted testing with continuous integration dashboards, code review processes, and architectural governance protocols. An advanced solution would autonomously propose tests, assess change risk, identify weak assertions shown by mutation testing, and alert developers to coverage deficiencies brought by new contributions. In that context, AI-enhanced TDD integrates into a whole intelligent quality-assurance ecosystem rather than functioning as a standalone automation initiative. From an organizational standpoint, incremental adoption is more feasible than a sudden overhaul of established testing methodologies.

An effective adoption strategy should commence with a pilot program focused on a somewhat comprehensible module instead of the most essential subsystem of a product. Throughout this pilot, the team will establish prompt templates, identify acceptable test patterns, and assess the frequency of AI-generated tests that are accepted with minimal modifications. This regulated access point diminishes opposition from developers and facilitates the generation of local proof supporting the scalability of the strategy.

The subsequent phase involves the establishment of a streamlined governance framework. Teams must delineate the repositories permissible for context, specify the project data that may be transmitted to external or internal AI services, establish protocols for prompt logging, and identify the individuals accountable for approving created tests in pull requests. These regulations convert AI help from an informal trial into a verifiable engineering discipline aligned with professional quality-assurance requirements.

The next step is integration with the current toolchain. Connecting AI-assisted test generation to version control hooks, static analysis, coverage reports, and continuous integration quality gates is recommended. The proposed tests are promptly evaluated for compilation, determinism, coverage contribution, and mutation resistance. If the produced artifacts fail these evaluations, they must revert to the refinement phase rather than being incorporated into the codebase unverified.

Ultimately, sustained acceptance necessitates ongoing calibration. Prompt tactics, rating models, and review criteria must be updated as the software develops, team conventions shift, and new categories of problems emerge in production. Consequently, AI-enhanced Test-Driven Development should be regarded as a regulated socio-technical process whereby technical automation, empirical feedback, and human accountability are closely intertwined.

An effective method to analyze the above arguments is to consider an exemplary quality scenario for the password-reset module in a more detailed manner. To clarify the discussion, consider a hypothetical scenario in which

the password-reset module is modified to incorporate enhanced anti-abuse measures and the translation of email templates. In a conventional procedure, developers must manually examine the altered code, ascertain which scenarios are impacted, and subsequently augment the relevant tests. The proposed AI-enhanced TDD approach entails converting requirement updates into structured constraints, enabling the test-generation component to promptly propose new negative and boundary scenarios, such as locale-specific template validation, restrictions on repeated requests, and combinations of expired or previously utilized tokens.

Assume that the risk model designates the token-validation service and rate-limiting middleware as the most susceptible components of the module due to their significant churn and a record of defect-fixing commits. Consequently, the testing effort can be focused on these places prior to the commencement of extensive regression execution. The developer can thereafter evaluate a more concise yet higher-value selection of possible tests, so enhancing the quality of the review process. This illustrates a fundamental concept of the article: AI not only produces additional tests but also assists in directing limited human attention to areas where testing is most likely to yield significant benefit.

A further advantage arises when mutation testing is incorporated into the same cycle. If several created tests execute the target code but fail to eliminate purposefully injected mutants, the system may categorize them as inadequate and return them for enhancement. Consequently, the approach becomes iterative at both the code and test quality levels. In business-critical modules like password recovery, it is crucial that tests not only exist but are also robust enough to identify logical flaws related to security, user experience, and availability limitations.

In an educational or research setting, the identical illustrated scenario can serve as a concise benchmark for evaluating quick techniques, LLM setups, or review protocols. The comprehensible business logic, along with its distinct edge situations, offers an ideal context for illustrating the impact of AI assistance on the Red-Green-Refactor cycle. The password-reset module serves as a useful micro-case for examining the interplay between automated help and developer judgment in modern software engineering, beyond merely exemplifying application logic.

## CONCLUSION

In order to enhance the planning, prioritization, and development of automated tests, the article has suggested a method for incorporating artificial intelligence into test-driven development. The approach combines requirement analysis, large-language-model-based test generation, machine-learning-based risk prioritization, and mutation-oriented optimization within the classical Red-Green-Refactor cycle.

The analysis has shown that the most valuable role of AI in TDD is not the automatic replacement of the developer, but the strengthening of developer decision-making through faster scenario discovery, broader coverage of edge cases, and more informed allocation of testing effort. The approach is especially promising for systems with complex business rules, frequent changes, and substantial verification overhead.

Further research should focus on benchmark-based evaluation of AI-assisted TDD in industrial projects, quality metrics for generated tests, and governance mechanisms that ensure transparency, reproducibility, and secure adoption of AI tools in software engineering practice.

## References

1. Beck K. Test-Driven Development: By Example. Boston: Addison-Wesley Professional, 2003. 240 p.
2. Janzen D. S., Saiedian H. Test-Driven Development: Concepts, Taxonomy, and Future Direction. Computer. 2005. Vol. 38, No. 9. P. 43-50. DOI: https://doi.org/10.1109/MC.2005.314
3. Cadar C., Godefroid P., Khurshid S., Pasareanu C. S., Sen K., Tillmann N., Visser W. Symbolic Execution for Software Testing in Practice: Preliminary Assessment. Proceedings of the 33rd International Conference on Software Engineering. 2011. P. 1066-1071. DOI: https://doi.org/10.1145/1985793.1985995
4. Utting M., Legeard B. Practical Model-Based Testing: A Tools Approach. San Francisco: Morgan Kaufmann, 2010. 680 p.
5. McMinn P. Search-Based Software Testing: Past, Present and Future. 2011 IEEE Fourth International Conference on Software Testing, Verification and Validation Workshops. 2011. P. 153-163. DOI: https://doi.org/10.1109/ICSTW.2011.100
6. Shivaji S., Whitehead E. J., Akella R., Kim S. Reducing Features to Improve Code Change-Based Bug Prediction. IEEE Transactions on Software Engineering. 2013. Vol. 39, No. 4. P. 552-569. DOI: https://doi.org/10.1109/TSE.2012.43
7. Carvalho G., Reboucas M., de Lucena C. J. P., Mota A., de Oliveira A. Test Case Generation from Natural Language Requirements Based on SCR Specifications. Proceedings of the 28th Annual ACM Symposium on Applied Computing. 2013. P. 1389-1394. DOI: https://doi.org/10.1145/2480362.2480591
8. Schafer M., Nadi S., Eghbali A., Tip F. An Empirical Evaluation of Using Large Language Models for Automated Unit Test Generation. IEEE Transactions on Software Engineering. 2024. Vol. 50. P. 85-105. DOI: https://doi.org/10.1109/TSE.2023.3334955
9. Yang L., Ye H., Wang H., Li Z., Zeng J., Jin Z., Jiang Y. On the Evaluation of Large Language Models in Unit Test Generation. Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering. 2024. P. 1607-1619. DOI: https://doi.org/10.1145/3691620.3695529
10. Jia Y., Harman M. An Analysis and Survey of the Development of Mutation Testing. IEEE Transactions on Software Engineering. 2011. Vol. 37, No. 5. P. 649-678. DOI: https://doi.org/10.1109/TSE.2010.62