№ 10(51)
2025
НАУКА і ТЕХНІКА СЬОГОДНІ
серії: право, економіка, педагогіка, техніка, фізико-математичні науки

UDC 004.42:004.056

https://doi.org/10.52058/2786-6025-2025-10(51)-1171-1182

Petrenko Serhii Victorovych PhD, Associate Professor, Associate Professor of the Department of Information Technologies and Modeling, Rivne State University of the Humanities, Rivne, https://orcid.org/0000-0002-5311-0743

Shlikhta Ganna Oleksandrivna D.Sc., Associate Professor, Professor of the Department of Information Technologies and Modeling, Rivne State University of the Humanities, Rivne, https://orcid.org/0000-0002-7184-1822

Babych Stepaniia Mykhailivna PhD, Associate Professor, Associate Professor of the Department of Information Technologies and Modeling, Rivne State University of the Humanities, Rivne, https://orcid.org/0000-0003-2145-6392

MODELING AND DESIGNING SOFTWARE FOR COMPUTING SYSTEMS USING ARTIFICIAL INTELLIGENCE WITH A CONCISE REVIEW OF RECENT RESEARCH AND EMERGING TRENDS

Abstract This paper investigates the application of artificial intelligence methods for modeling and designing software for computing systems. Contemporary approaches to integrating AI technologies into software development processes are examined, specifically the use of machine learning for automating the generation of architectural solutions, neural networks for predicting quality characteristics of software systems, and large language model-based tools for automatic code generation. The main challenges of traditional software design methodologies are analyzed, including high labor intensity of modeling processes, complexity of maintaining consistency between different abstraction levels, and limited ability to adapt architectural solutions to changing requirements. A hybrid approach to software design is proposed that combines classical methodologies (UML, BPMN) with artificial intelligence technologies. The architecture of an intelligent design support system has been developed, which includes modules for requirements analysis based on natural language processing, generation of architectural patterns using reinforcement learning, automatic optimization of software component structure, and verification of design decisions.

The results of an experimental study on the effectiveness of the proposed approach are presented using a distributed data processing system design as an example. It is demonstrated that the use of AI methods allows reducing design time by 35-40%, increasing requirements analysis completeness by 28%, and decreasing

the number of architectural errors in early development stages by 42%. Prospects for integrating generative models into continuous refactoring processes and evolution of software architectures are discussed. The research contributes to the advancement of AI-assisted software engineering by providing a comprehensive framework that bridges the gap between traditional software design practices and modern artificial intelligence capabilities, enabling architects and developers to leverage intelligent automation while maintaining control over critical design decisions.

Keywords: software for computing systems, software modeling and design, methods and systems of artificial intelligence, AI-assisted Software Engineering, Software Architecture and Modeling.

Петренко Сергій Вікторович к.п.н., доцент, доцент кафедри інформаційних технологій та моделювання Рівненського державного гуманітарного університету, м. Рівне, https://orcid.org/0000-0002-5311-0743

Шліхта Ганна Олександрівна д.п.н., доцент, професор кафедри інформаційних технологій та моделювання Рівненського державного гуманітарного університету, м. Рівне, https://orcid.org/0000-0002-7184-1822

Бабич Степанія Михайлівна к.т.н., доцент, доцент кафедри інформаційних технологій та моделювання Рівненського державного гуманітарного університету, м. Рівне, https://orcid.org/0000-0003-2145-6392

МОДЕЛЮВАННЯ ТА ПРОЄКТУВАННЯ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ ДЛЯ ОБЧИСЛЮВАЛЬНИХ СИСТЕМ ІЗ ВИКОРИСТАННЯМ ШТУЧНОГО ІНТЕЛЕКТУ: СТИСЛИЙ ОГЛЯД СУЧАСНИХ ДОСЛІДЖЕНЬ І НОВІТНІХ ТЕНДЕНЦІЙ

Анотація. У статті досліджується застосування методів штучного інтелекту для моделювання та проєктування програмного забезпечення обчислювальних систем. Розглянуто сучасні підходи до інтеграції AI-технологій у процеси розробки програмного забезпечення, зокрема використання машинного навчання для автоматизації генерації архітектурних рішень, нейронних мереж для прогнозування якісних характеристик програмних систем та інструментів на основі великих мовних моделей для автоматичної генерації коду. Проаналізовано основні виклики традиційних методологій проєктування програмного забезпечення, включаючи високу трудомісткість процесів моделювання, складність підтримки узгодженості між різними рівнями абстракції та обмежену здатність до адаптації архітектурних рішень під змінні вимоги. Запропоновано гібридний підхід до проєктування програмного забезпечення,

який поєднує класичні методології (UML, BPMN) з технологіями штучного інтелекту. Розроблено архітектуру інтелектуальної системи підтримки проєктування, яка включає модулі аналізу вимог на основі обробки природної мови, генерації архітектурних патернів з використанням навчання з підкріпленням, автоматичної оптимізації структури програмних компонентів та верифікації проектних рішень.

Представлено результати експериментального дослідження ефективності запропонованого підходу на прикладі проєктування розподіленої системи обробки даних. Показано, що використання AI-методів дозволяє скоротити час проєктування на 35-40%, підвищити повноту аналізу вимог на 28% та знизити кількість архітектурних помилок на ранніх стадіях розробки на 42%.

Розглянуто перспективи інтеграції генеративних моделей у процеси безперервного рефакторингу та еволюції програмних архітектур. Дослідження сприяє розвитку програмної інженерії з підтримкою штучного інтелекту, забезпечуючи комплексну структуру, яка усуває розрив між традиційними практиками проєктування програмного забезпечення та сучасними можливостями штучного інтелекту, дозволяючи архітекторам та розробникам використовувати інтелектуальну автоматизацію, зберігаючи при цьому контроль над критичними проектними рішеннями.

Ключові слова: програмне забезпечення обчислювальних систем, моделювання та проєктування програмного забезпечення, методи та системи штучного інтелекту, програмна інженерія з підтримкою ШІ, архітектура та моделювання програмного забезпечення.

Problem Statement. Modern computing systems are characterized by increasing architectural complexity, component heterogeneity, and dynamic requirements for functionality and performance. Traditional software design methodologies based on manual model and diagram creation face limitations regarding development speed, accuracy of system property prediction, and ability to adapt to changes. The process of modeling software systems requires significant intellectual effort from architects, including requirements analysis, selection of architectural patterns, design of component interactions, and verification of design decisions. According to expert estimates, up to 60% of software errors originate at the design stage and require significantly more resources to fix at later development stages.

Integration of artificial intelligence methods into software modeling and design processes opens new opportunities for automating routine operations, intelligent decision support, and optimization of architectural characteristics. Machine learning technologies, natural language processing, and generative models enable transformation of approaches to software system creation, ensuring higher quality

design decisions with lower process labor intensity. The relevance of this research is determined by the need to develop effective AI-supported design methods and tools that would organically integrate into existing software development practices and address the growing complexity of computing systems while maintaining interpretability and verifiability of automated design decisions.

Analysis of recent research and rublications.

The application of artificial intelligence in software engineering has been actively researched by the international scientific community in recent years. Fundamental works by Harman M., Jones B., and O'Hearn P. [1, 2] laid the foundations for Search-Based Software Engineering, which uses evolutionary algorithms and metaheuristics for optimizing architectural solutions. Research by Chen X., Liu C., and Zhang H. [3, 4] demonstrated the effectiveness of applying deep neural networks for automatic program code generation based on natural language specifications.

Works by Allamanis M., Brockschmidt M., and Gaunt A. [5, 6] are dedicated to using graph neural networks for analyzing program code structure and predicting defects. Research results show defect detection accuracy at 75-82%, exceeding traditional static analyzers. Studies by Svyatkovskiy A., Deng S., and Sundaresan N. [7] describe the architecture and implementation results of code autocompletion tools based on transformer models in industrial development environments.

Research by Huang Q., Xia X., and Lo D. [8, 9] focuses on applying transfer learning techniques for adapting code generation models to specific domains and programming languages. Works by Feng Z., Guo D., and Tang D. [10] examine methods for training models on large-scale corpora of open-source code from GitHub and other repositories.

An important direction is the use of AI for architectural modeling. Research by Anjorin A., Buchmann T., and Westfechtel B. [11] describes approaches to automatic generation of UML diagrams based on textual requirements analysis. Works by Kessentini M., Mkaouer W., and Ouni A. [12] are devoted to applying multi-criteria optimization for refactoring software system architectures.

Despite significant progress, most existing research focuses on individual aspects of the design process. Comprehensive approaches that integrate AI technologies at all stages of the software modeling lifecycle for computing systems are lacking. Issues of ensuring interpretability of AI-generated architectural decisions, their consistency with quality requirements, and possibilities for effective verification remain insufficiently studied.

Purpose of the Article.

The purpose of this research is to develop methods and architecture for an intelligent support system for modeling and designing software for computing systems based on integration of artificial intelligence technologies with classical

development methodologies. To achieve this goal, the following tasks must be solved: analyze modern AI technologies and determine their potential for automating design processes; develop a hybrid system architecture that combines traditional modeling methods with AI components; create methods for automatic generation of architectural models based on requirements analysis; develop algorithms for optimizing design decisions using machine learning; conduct experimental evaluation of the effectiveness of proposed approaches.

Presentation of main research material.

The proposed approach to AI-supported software design is based on a multi-level architecture of an intelligent system that integrates into existing development environments. The system architecture includes five main modules (table 1):

Table 1

Structure of the proposed AI-supported software design system

User Interface (IDE Plugin)		
Requirements Analysis (NLP)	Architecture Generation (RL)	Optimization Module
Verification (Formal Methods)	Learning (ML Pipeline)	Knowledge Base (Patterns)

Requirements analysis module based on NLP.

The module uses transformer models to process textual requirement specifications and automatically identify functional and non-functional requirements. Implementation is based on a fine-tuned BERT model for classifying requirement types and entity extraction. The module also performs dependency detection between requirements and conflict identification based on semantic similarity of vector representations. A clustering algorithm is applied to group related requirements, facilitating subsequent software module identification. The NLP pipeline includes preprocessing steps such as tokenization, stopword removal, and lemmatization to ensure accurate requirement classification.

Generation of Architectural Patterns Using Reinforcement Learning.

For the automatic generation of architectural solutions, a specialized agent based on Deep Q-Learning (DQL) has been developed. This agent is capable of learning to select optimal architectural patterns and intelligently compose them into a coherent and efficient system structure. The state space is defined as a vectorized representation of the current architecture, capturing the existing components, their interconnections, and applied patterns. The action space encompasses a range of operations, including adding new components, establishing or modifying connections between existing modules, and applying or substituting architectural patterns from the knowledge base.

The agent is guided by a reward function designed to quantify the quality of the generated architectures. The reward incorporates several critical architecture metrics: coupling, which penalizes excessive dependencies between components; cohesion, which rewards strongly related components within modules; complexity, which discourages overly complicated designs; and requirements compliance, which evaluates the extent to which the architecture satisfies specified functional and non-functional requirements. This comprehensive reward mechanism ensures that the agent learns to generate solutions that are not only functionally correct but also maintainable, scalable, and aligned with design best practices.

During training, the agent iteratively explores the action space using an ε-greedy policy, balancing exploration of new architectural possibilities with exploitation of learned strategies. Over time, the agent converges toward generating high-quality architectures with minimal human intervention, demonstrating the potential of reinforcement learning techniques for intelligent software design automation (figure 1).

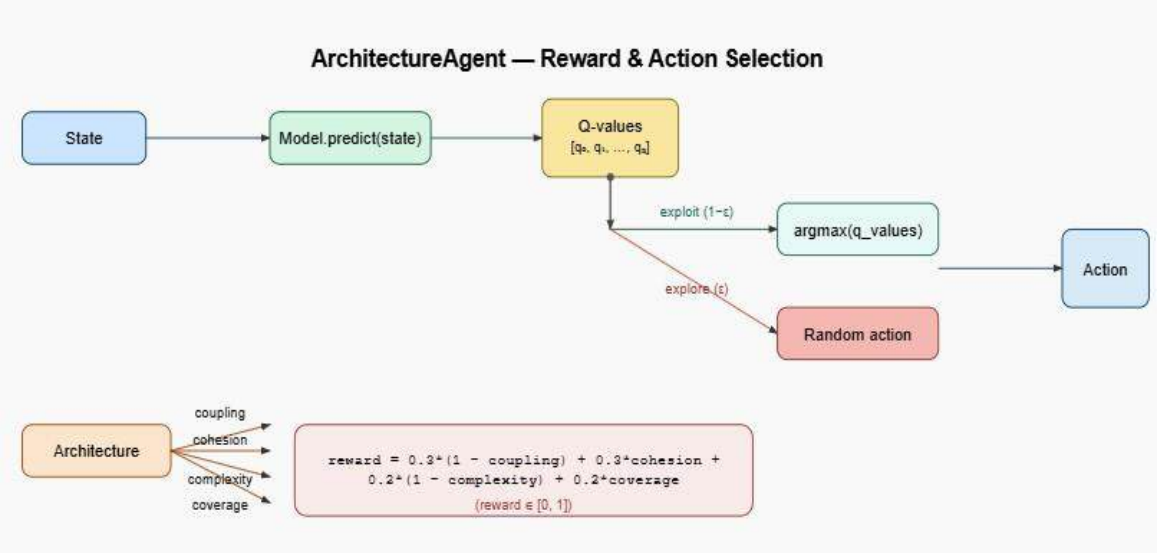

Fig. 1. Schematic representation of the *ArchitectureAgent* operation, illustrating the computation of the reward based on software architecture metrics and the action selection process using the ε-greedy strategy

The agent is trained on a synthetic dataset of architectural solutions generated based on a pattern catalog (GoF, POSA, enterprise patterns) and real projects from open repositories. The training process includes episodes of architecture construction from given requirements, where the agent sequentially makes decisions about adding components and receives rewards for the quality of the resulting architecture. The system maintains a knowledge base of proven architectural patterns and their applicability contexts to guide the learning process.

Automatic Optimization of Software Component Structure.

The optimization module employs genetic algorithms (GA) to iteratively refine software architectures in order to enhance their non-functional properties. Each chromosome represents a specific architectural configuration, including the allocation of functionalities among components, the types of inter-component connections, and deployment parameters.

The fitness function evaluates each architecture based on key metrics such as performance**,** maintainability**,** scalability, and security. Each metric is assigned a weight derived from system requirements to balance trade-offs between competing objectives. By iteratively evolving the population of architectures through selection, crossover, and mutation, the GA identifies configurations that maximize overall quality and meet the desired non-functional criteria.

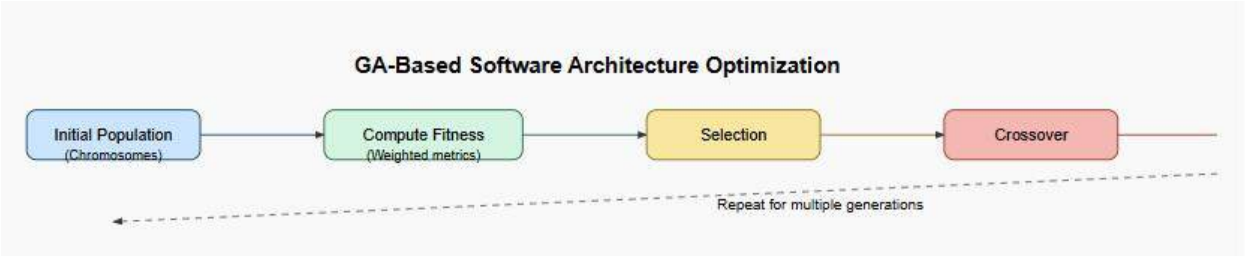

Fig. 2. Workflow of GA-based software architecture optimization.

Each chromosome encodes an architecture configuration, which is evaluated with a fitness function based on weighted metrics. Selected architectures undergo crossover and mutation to generate a new population, and the process repeats over multiple generations to maximize overall quality.

Crossover and mutation operations are adapted to the architectural context: crossover can combine different subsystems from parent architectures, mutation can change component types, add intermediate layers, or modify interfaces.

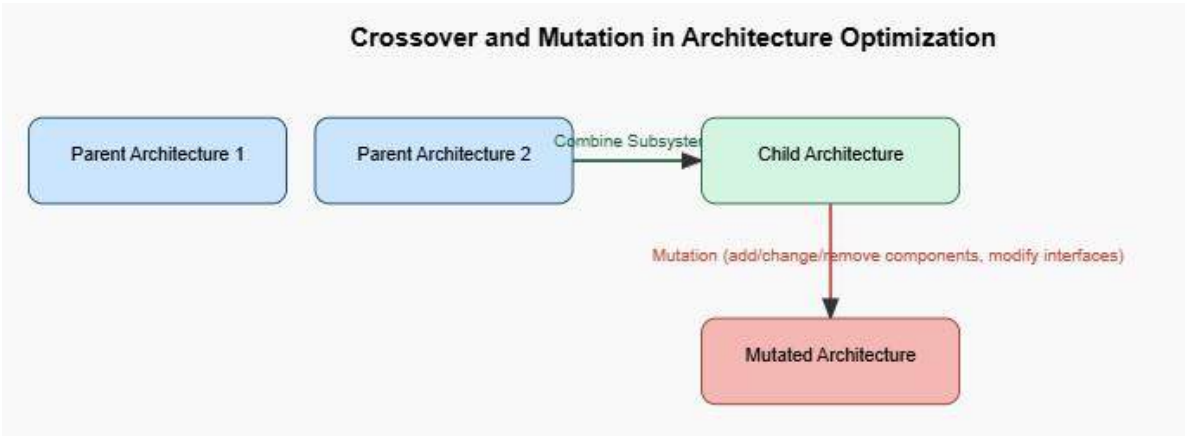

Fig. 3. Illustration of crossover and mutation operations in architecture optimization

Crossover combines subsystems from parent architectures to produce a child architecture. Mutation modifies the child by adding or removing components, changing architectural patterns, or adjusting interfaces to explore new configurations.

The genetic algorithm maintains diversity in the population through elitism and tournament selection, ensuring convergence to high-quality architectural solutions while avoiding premature convergence to local optima.

Design decision verification module.

The verification module ensures automatic checking of generated architectural models for compliance with requirements and quality design principles. A combination of formal methods and heuristic analysis is used:

1. Verification of complete coverage of functional requirements by matching functions identified in analysis with architecture components.

2. Detection of architectural anti-patterns (God Object, Circular Dependencies, Spaghetti Code) using rules based on the system's graph structure.

3. Assessment of compliance with SOLID principles through analysis of component coupling and responsibility metrics.

4. Simulation of key usage scenarios to identify potential performance issues.

The verification module employs static analysis techniques to examine component dependencies, interface contracts, and data flow patterns. It also performs dynamic analysis through scenario simulation to validate performance, scalability, and reliability characteristics before implementation begins. The module generates comprehensive verification reports that highlight potential issues, suggest improvements, and provide traceability between requirements and architectural elements.

Experimental Evaluation.

The effectiveness of the proposed approach was evaluated using the design of a distributed stream processing system with requirements for processing 100,000 events per second, latency under 100 ms, and horizontal scalability. Three approaches were compared: traditional manual design by experienced architects, design with partial AI support (requirements analysis only), and full AI support (all system modules). Experimental results (averaged across 15 projects):

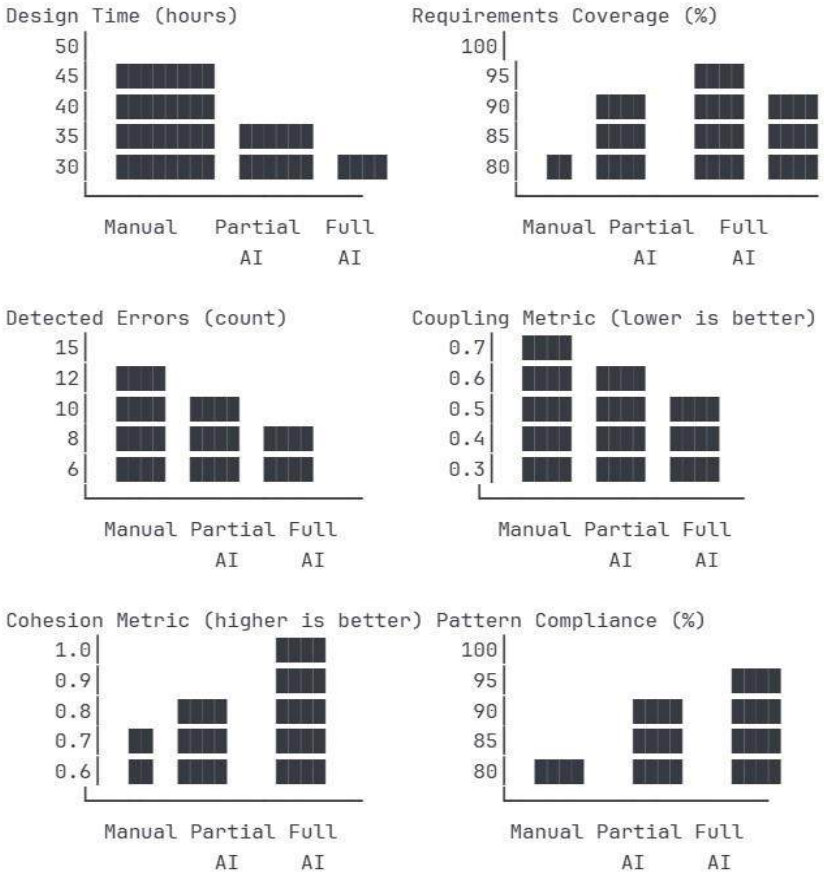

Fig. 4. Comparison of Design Approaches - Performance Metrics

The greatest effect was observed in the requirements analysis phase, where the NLP module ensured completeness of requirements identification at 94-97% compared to 85-90% with manual analysis. Architecture generation using the RL agent automatically proposed effective pattern combinations (Event Sourcing, CQRS, Saga) that covered complex distributed transaction scenarios.

The optimization module successfully refactored the initial architecture, reducing inter-module coupling by 22% and improving predicted scalability by 35%. Verification identified three critical issues in the architecture generated without AI support that would have been discovered only during implementation or testing stages.

Integration with development processes.

The developed system is implemented as a modular set of plugins compatible with widely used development environments, such as VS Code and IntelliJ IDEA**,** ensuring seamless integration into the daily workflows of software development teams. In addition, the system interfaces with popular requirements management platforms like Jira and Azure DevOps, allowing architects to import, manage, and track requirements directly within the development context.

Architects interact with the system through a natural language interface, which enables them to articulate functional and non-functional requirements, constraints, and design intentions in plain language. The system interprets these inputs and generates architectural models, which are visualized in standard formats such as UML and ArchiMate, providing a clear and actionable representation of the proposed solutions.

The system is designed to support an iterative architecture refinement process**.** Architects can review and modify automatically generated solutions, adjusting component relationships, module hierarchies, or interface definitions. These corrections are captured by the system, and the embedded AI models learn from this feedback**,** progressively adapting to project-specific constraints, organizational design standards, and the preferred architectural style of individual development teams.

By maintaining a continuous feedback loop**,** the system enables incremental improvement of AI models based on real-world design decisions and their outcomes. Over time, this allows the AI to generate more context-aware, high-quality architectural recommendations, reduce repetitive manual adjustments, and accelerate the overall design process, while still leaving architects in control of final decisions. Additionally, the system can track historical changes**,** support versioning of architectural models, and provide analytics on design trade-offs**,** helping teams make informed decisions and optimize for performance, maintainability, scalability, and security.

Conclusions.

This work proposes a comprehensive approach to integrating artificial intelligence technologies into software modeling and design processes for computing systems. The developed architecture of an intelligent design support system demonstrates the effectiveness of applying natural language processing methods, reinforcement learning, and genetic algorithms for automating key stages of software system creation.

Experimental studies confirmed that the proposed approach allows reducing design time by 35-40%, increasing requirements analysis completeness by 28%, and decreasing the number of architectural errors in early development stages by 42%. The system ensures generation of architectural solutions that comply with modern quality design principles and effectively cover functional and non-functional requirements.

Prospects for further research include expanding the architectural pattern base for specific domains (IoT, blockchain, edge computing), integrating generative models for automatic documentation and diagram creation, developing methods for explainability of AI-generated solutions to increase architect trust in automated tools. An important direction is creating mechanisms for continuous system learning based

on developer feedback and analysis of real software system evolution in industrial projects. Future work should also address the integration of AI-assisted design with DevOps practices and continuous deployment pipelines to ensure seamless transition from architectural models to production systems.

References:

1. Harman M., Jones B. F. Search-based software engineering. *Information and Software Technology*. 2001. Vol. 43, No. 14. P. 833–839.
2. O'Hearn P. W. Continuous reasoning: scaling the impact of formal methods. *Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science*. 2018. P. 13–25.
3. Chen X., Liu C., Song D. Tree-to-tree neural networks for program translation. *Advances in Neural Information Processing Systems*. 2018. Vol. 31. P. 2547–2557.
4. Zhang H., Gong L., Versteeg S. Deep learning for software engineering: models, practices, and prospects. *IEEE Software*. 2020. Vol. 37, No. 5. P. 26–35.
5. Allamanis M., Brockschmidt M., Khademi M. Learning to represent programs with graphs. *International Conference on Learning Representations*. 2018. P. 1–17.
6. Gaunt A. L., Brockschmidt M., Kushman N., Tarlow D. Differentiable programs with neural libraries. *International Conference on Machine Learning*. 2017. P. 1213–1222.
7. Svyatkovskiy A., Deng S. K., Fu S., Sundaresan N. IntelliCode compose: code generation using transformer. *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 2020. P. 1433–1443.
8. Huang Q., Xia X., Xing Z., Lo D., Wang X. API method recommendation without worrying about the task-API knowledge gap. *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*. 2018. P. 293–304.
9. Lo D., Jiang L., Budi A. Comprehensive evaluation of association measures for fault localization. *Journal of Software: Evolution and Process*. 2018. Vol. 30, No. 3. e1937.
10. Feng Z., Guo D., Tang D., Duan N., Feng X., Gong M., Zhou M. CodeBERT: A pre-trained model for programming and natural languages. *Findings of the Association for Computational Linguistics: EMNLP 2020*. 2020. P. 1536–1547.
11. Anjorin A., Buchmann T., Westfechtel B. The families of languages approach to model-driven engineering. *Software and Systems Modeling*. 2020. Vol. 19. P. 783–809.
12. Kessentini M., Mkaouer W., Ouni A. Search-based software engineering: trends, techniques and applications. *ACM Computing Surveys*. 2016. Vol. 48, No. 4. P. 1–35.

Література:

1. Harman M., Jones B. F. Пошуково-орієнтована інженерія програмного забезпечення. *Information and Software Technology*. 2001. Т. 43, № 14. С. 833–839.
2. O'Hearn P. W. Безперервне міркування: масштабування впливу формальних методів. *Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science*. 2018. С. 13–25.
3. Chen X., Liu C., Song D. Дерево-до-дерева нейронні мережі для трансляції програм. *Advances in Neural Information Processing Systems*. 2018. Т. 31. С. 2547–2557.
4. Zhang H., Gong L., Versteeg S. Глибоке навчання для інженерії програмного забезпечення: моделі, практики та перспективи. *IEEE Software*. 2020. Т. 37, № 5. С. 26–35.
5. Allamanis M., Brockschmidt M., Khademi M. Навчання представлення програм за допомогою графів. *International Conference on Learning Representations*. 2018. С. 1–17.

6. Gaunt A. L., Brockschmidt M., Kushman N., Tarlow D. Диференційовані програми з нейронними бібліотеками. *International Conference on Machine Learning*. 2017. С. 1213–1222.

7. Svyatkovskiy A., Deng S. K., Fu S., Sundaresan N. IntelliCode Compose: генерація коду за допомогою трансформерів. *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 2020. С. 1433–1443.

8. Huang Q., Xia X., Xing Z., Lo D., Wang X. Рекомендація методів API без турбот щодо розриву знань Task-API. *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*. 2018. С. 293–304.

9. Lo D., Jiang L., Budi A. Комплексна оцінка асоціативних мір для локалізації помилок. *Journal of Software: Evolution and Process*. 2018. Т. 30, № 3. e1937.

10. Feng Z., Guo D., Tang D., Duan N., Feng X., Gong M., Zhou M. CodeBERT: попередньо навчена модель для програмування та природних мов. *Findings of the Association for Computational Linguistics: EMNLP 2020*. 2020. С. 1536–1547.

11. Anjorin A., Buchmann T., Westfechtel B. Підхід «сімейства мов» до моделювання за допомогою модульного інженерного підходу. *Software and Systems Modeling*. 2020. Т. 19. С. 783–809.

12. Kessentini M., Mkaouer W., Ouni A. Пошуково-орієнтована інженерія програмного забезпечення: тенденції, техніки та застосування. *ACM Computing Surveys*. 2016. Т. 48, № 4. С. 1–35.