DOI: 10.58254/viti.7.2025.08.81

УДК 004.05

канд. техн. наук, доцент Любарський С. В. ORCID: 0000-0001-8068-1106 (ВІТІ ім. Героїв Крут)
Кондратюк А. Г. ORCID: 0000-0002-8485-3160 (ВІТІ ім. Героїв Крут)
Шарнін С. А. ORCID: 0000-0002-5856-3995 (ВІТІ ім. Героїв Крут)
канд. техн. наук, доцент Здоренко Ю. М. ORCID: 0000-0002-5649-771X (НУ «Полтавська політехніка ім. Ю. Кондратюка»)

ФУНКЦІОНАЛЬНА МОДЕЛЬ РЕФАКТОРИНГУ ОБ'ЄКТНО-ОРІЄНТОВАНОГО КОДУ НА ОСНОВІ МЕТОДІВ ШТУЧНОГО ІНТЕЛЕКТУ ДЛЯ ЗАБЕЗПЕЧЕННЯ ЗАВДАННЯ БЕЗПЕКИ ПРОГРАМНОГО КОДУ

Сучасне програмне забезпечення задає високі вимоги до безпеки та якості коду. Одним із важливих завдань безпечного використання сучасних інформаційних систем є запровадження заходів, що унеможливлюють спроби через існуючи вразливості програмного забезпечення впливати на функціональність цих систем. Особливо важлива інтеграція механізмів дотримання безпеки та якості програмного коду у всі фази життєвого циклу проєктування інформаційної системи. Найбільш перспективним напрямом у вирішенні цього завдання виступає застосування процесу рефакторингу програмного коду.

Існуючі підходи до рефакторингу мають чисельні обмеження. Методи штучного інтелекту можуть значно допомогти подолати ці обмеження, надаючи автоматизовані, об'єктивні та адаптивні рішення для покращення якості, безпеки та підтримуваності програмного коду.

Використання методів штучного інтелекту дозволяє автоматизувати процес статичного та динамічного аналізу програмного коду, виявити проблеми і вирішити їх шляхом пропозиції оптимізовано-безпечного та ефективного варіанту.

Пропонується до розгляду модель рефакторингу об'єктно-орієнтованого коду на основі методів штучного інтелекту з метою підвищення рівня його якості та безпеки.

Ключові слова: *функціональна модель, програмне забезпечення, статичний і динамічний аналіз, рефакторинг, технології штучного інтелекту, вразливості безпеки, якість коду.*

S. Liubarskyi, A. Kondratiuk, S. Sharnin, Y. Zdorenko. Functional model of object-oriented code refactoring based on artificial intelligence methods to provide security tasks of software code

Modern software sets high requirements for code security and quality. One of the important tasks of safe use of modern information systems is to implement measures that make it impossible to attempt to influence the functionality of these systems through existing software vulnerabilities. Integration of mechanisms for maintaining security and quality of program code into all phases of the information system design life cycle is especially important. The most promising direction in solving this problem is the use of the program code refactoring process.

Existing approaches to refactoring have numerous limitations. Artificial intelligence methods can significantly help overcome these limitations by providing automated, objective and adaptive solutions to improve the quality, security and maintainability of program code.

The use of artificial intelligence methods allows you to automate the process of static and dynamic analysis of program code, identify problems and solve them by offering an optimized, safe and effective option.

A model for refactoring object-oriented code based on artificial intelligence methods is proposed for consideration in order to improve its quality and security.

Keywords: *functional model, software, static and dynamic analysis, refactoring, artificial intelligence technologies, security vulnerabilities, code quality.*

Постановка завдання. Сучасне програмне забезпечення стикається з дедалі більшими вимогами до безпеки та якості програмного коду через зростаючий рівень кібернетичних загроз, збільшення обсягу даних, що підлягають обробці, і високі очікування користувачів від програмного забезпечення, особливо в контурах підтримки прийняття рішень.

Безпека – це ключовий аспект сучасного програмного забезпечення, оскільки вразливості можуть призвести до серйозних наслідків, таких як витік даних, фінансові збитки або порушення конфіденційності. Якість коду забезпечує надійність, підтримуваність і масштабованість програмного забезпечення. Це також допомагає мінімізувати кількість

помилок і полегшити співпрацю команди розробників по досягненню цільової настанови програмної розробки.

Головними важелями, що потенційно впливають на безпеку та якість коду, є:

– передбачення найбільш документованих вразливостей, виявлення потенційних загроз у програмному об'єктно-орієнтованому коді за допомогою методів статичного та динамічного аналізу;

– інтеграція у вихідний код механізмів обробки виняткових ситуацій без переривання роботи програми, використання механізмів відкату;

– забезпечення читабельності та структурованості коду;

– розбиття коду на маленькі, незалежні модулі або функції, використання принципу *Don't Repeat Yourself*;

– використання ефективних алгоритмів і структур даних, уникнення надмірного споживання ресурсів (*CPU*, *RAM*, мережевої пропускної здатності);

– використання систем контролю версій (наприклад, *Git*) для управління змінами програмного коду, дотримання практик *Code Review* для забезпечення якості;

– дотримання рекомендацій *OWASP* для безпечного програмування, використання сучасних методологій розробки (*Agile*, *DevOps*);

– написання модульних тестів (*unit tests*), інтеграційних тестів і тестів регресії, використання інструментів автоматизації тестування (наприклад, *JUnit*, *pytest*).

Для забезпечення високих стандартів безпеки та якості, особливо для завдань проєктування, розробки і супроводження інформаційних систем спеціального призначення, важливо інтегрувати механізми їхньої реалізації у всі етапи життєвого циклу програмного забезпечення.

Найбільш перспективним напрямом у вирішенні цього завдання виступає застосування процесу рефакторингу програмного коду.

Рефакторинг програмного коду – це процес внесення змін у існуючий програмний код з метою покращення його структури, читабельності та підтримуваності без зміни його зовнішньої поведінки. Рефакторинг є невід'ємною частиною розробки програмного забезпечення, оскільки він допомагає підтримувати якість програмного коду на високому рівні і може значно впливати на його безпеку. У контексті безпеки рефакторинг часто стає превентивною мірою, яка запобігає потенційним атакам та зменшує ризики.

Існуючі підходи до рефакторингу мають чисельні обмеження, такі як трудомісткість, суб'єктивність, відсутність глибокого розуміння контексту, недостатня масштабованість та адаптивність. Методи штучного інтелекту можуть значно допомогти подолати ці обмеження, надаючи автоматизовані, об'єктивні та адаптивні рішення для покращення якості, безпеки та підтримуваності програмного коду.

Використання методів штучного інтелекту (ШІ) дозволить автоматизувати процес аналізу, виявлення проблем та їх усунення [1]. Пропонується для розгляду модель рефакторингу об'єктно-орієнтованого програмного коду на основі методів ШІ для підвищення рівня його безпеки та якості.

Аналіз останніх досліджень і публікацій. Основні вразливості, що потенційно впливають на безпеку та якість програмного коду, представлено на рисунку 1 [2].

SQL-ін'єкції є однією з найнебезпечніших вразливостей. Окрім компрометуючих впливів (доступ до критично важливих даних, зміна структури даних або навіть руйнація їх вмісту), зниження продуктивності інформаційної системи шляхом виконання неконтрольованих запитів, *SQL*-ін'єкції можуть впливати на якість коду. Використання некоректних або незахищених запитів у коді ускладнює його підтримку, підвищує ризик появи нових помилок і вразливостей.

Методи запобігання *SQL*-ін’єкціям включають використання підготовлених запитів (*prepared statements*), *ORM* (*Object-Relational Mapping*) для взаємодії з базами даних, а також застосування механізмів валідації та фільтрації введених даних.

XSS-атаки (*Cross-Site Scripting*) можуть викликати маніпуляції з контентом вебдодатка шляхом уведення дезінформації або підробки елементів інтерфейсу. Це призводить до зниження довіри до додатка, погіршення завдання підтримуваності коду. Недостатня перевірка введених даних, невикористання механізмів фільтрації сприяє накопиченню вразливостей у коді [3; 4].

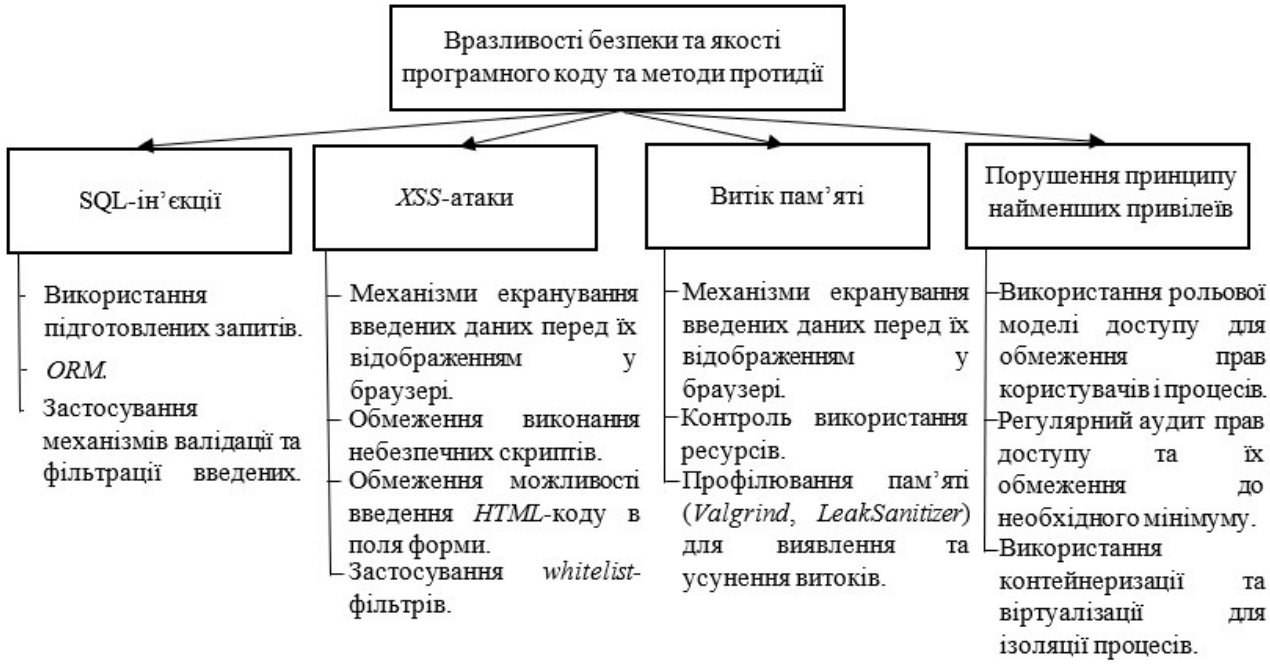

Рис. 1. Основні вразливості безпеки та якості програмного коду та методи їхньому запобіганню

Запобігти у цьому можуть механізми екранування (*escaping*) введених даних перед їх відображенням у браузері, впровадження *CSP* (*Content Security Policy*) для обмеження виконання небезпечних скриптів, обмеження можливості введення *HTML*-коду в поля форми, застосування *whitelist*-фільтрів.

Витік пам’яті (*memory leak*) виникає, коли програмна система неправильно звільняє використані ресурси, що може призвести до поступового зростання споживання пам’яті та збою програмної системи.

Основними наслідками цього може бути: зниження продуктивності програмного коду (неефективне використання пам’яті призводить до уповільнення роботи додатка через збільшене навантаження на оперативну пам’ять); підвищений ризик відмови програмної системи (у критичних системах витік пам’яті може призвести до повного вичерпання доступних ресурсів і аварійного завершення роботи програмної системи); уразливість до *DoS*-атак (зловмисники можуть свідомо спричиняти витоки пам’яті для виснаження ресурсів сервера); погіршення якості програмного коду (код, схильний до витоків пам’яті, є складним у підтримці та модернізації).

Методами, що можуть запобігати цій загрозі, є: використання автоматичного керування пам’яттю (*garbage collection*) у мовах програмування, які це підтримують; контроль використання ресурсів, зокрема явне звільнення пам’яті у мовах без автоматичного

управління пам'яттю (наприклад, *C#*, *C++*); використання інструментів для профілювання пам'яті (*Valgrind, LeakSanitizer*), виявлення та усунення витоків.

Порушення принципу найменших привілеїв відбувається, коли користувачі або процеси отримують доступ до ресурсів або функцій, які їм не потрібні для виконання своїх завдань. Насамперед це впливає на несанкціонований доступ до критичних даних. Надання надмірних привілеїв у коді ускладнює аудит безпеки та підвищує ризик появи несанкціонованих змін.

Методи запобігання порушенню принципу найменших привілеїв включають: використання рольової моделі доступу (*RBAC*) для обмеження прав користувачів і процесів; регулярний аудит прав доступу та їх обмеження до необхідного мінімуму; використання контейнеризації та віртуалізації для ізоляції процесів.

Виявлення потенційних загроз у програмному об'єктно-орієнтованому коді здійснюється за допомогою статичного та динамічного аналізу програмного коду.

Методи статичного аналізу (*Static Code Analysis*) спрямовані на дослідження програмного коду без його виконання. Основна мета статичного аналізу – виявлення вразливостей, помилок програмування, порушень стилю кодування та недоліків архітектури. Методи цієї групи найбільш оптимальні для виявлення вразливостей на ранніх етапах розробки (наприклад, *SQL*-ін'єкції, *XSS*, неправильне управління пам'яттю), перевірки дотримання стандартів кодування, аналізу залежностей та складності коду, автоматизованого рефакторингу та пошуку дублікатів коду.

Методи статичного аналізу покладаються на: використання інструментів аналізу коду (*SonarQube, ESLint, Pylint, Checkmarx*); використання процедур символічного виконання (*Symbolic Execution*); аналіз потоку даних (*Data Flow Analysis*).

На противагу статичному аналізу, динамічний аналіз (*Dynamic Code Analysis*) – це метод перевірки програмного коду під час його виконання. Він дозволяє виявити потенційні вразливості, що виникають лише під час реальної роботи програмної системи. Серед основних аспектів динамічного аналізу можна визначити: тестування безпеки під час виконання (*Runtime Security Testing*); виявлення витоків пам'яті, переповнення буфера, некоректного використання ресурсів; функціональне та навантажувальне тестування для оцінки продуктивності; аналіз поведінки програми за допомогою інструментів профілювання (*Valgrind, Dynatrace*, *AppDynamics*).

Динамічний аналіз дозволяє знаходити помилки, які не можуть бути виявлені за допомогою статичного аналізу, наприклад: вразливості, що виникають під певними умовами виконання програмного коду; атаки типу *race condition*; проблеми з конкурентним доступом до пам'яті.

Обидва методи аналізу є взаємодоповнюючими: статичний аналіз ефективний для раннього виявлення проблем, а динамічний – для тестування реальної поведінки програмного коду.

Статичний і динамічний методи аналізу програмного коду є основою для використання методів ШІ у рефакторингу. Вони дозволяють зібрати необхідні дані для навчання моделей машинного навчання та створення ефективних алгоритмів автоматизованого покращення коду.

Статичний аналіз надає інформацію про синтаксичні, стилістичні та структурні проблеми об'єктно-орієнтованого програмного коду, що використовується для навчання моделей ШІ та створення рекомендацій щодо рефакторингу [5].

Динамічний аналіз виявляє поведінкові проблеми програмного коду, такі як витоки пам'яті та проблеми продуктивності, що дозволяє адаптивно застосовувати методи оптимізації.

Метою статті є проведення порівняльного аналізу методів машинного та глибокого навчання технологій ШІ для автоматизації процесу рефакторингу програмного об'єктно-орієнтованого коду і на їх основі пропозиції функціональної моделі цього процесу в завданнях розробки надійного та якісного програмного коду інформаційних систем спеціального призначення.

Виклад основного матеріалу дослідження. Найбільшого застосування для автоматизації процесу рефакторингу програмного об'єктно-орієнтованого коду набули методи машинного та глибокого навчання технологій ШІ (рис. 2).

Як найбільш перспективні підходи слід розглядати:

1. *Використання нейронних мереж для аналізу коду та пошуку патернів "поганого" коду* (*code smells*). Нейронні мережі можуть ефективно аналізувати великі обсяги програмного коду та виявляти шаблони, що можуть свідчити про проблеми з безпекою та якістю.

Рис. 2. Методи машинного та глибокого навчання технологій ШІ для автоматизації процесу рефакторингу програмного коду

Основними напрямками їхнього застосування можна вважати: глибинне навчання для розпізнавання патернів коду (наприклад, навчання на великих наборах даних коду для виявлення поганих практик програмування, використання рекурентних нейронних мереж та трансформерів для аналізу послідовності коду); визначення "антипатернів" програмування (виявлення надмірної складності коду, дублікатів, порушень принципів *SOLID*, ідентифікація небезпечних конструкцій (наприклад, використання *eval*() або неконтрольованих введень користувача); автоматизований рефакторинг за допомогою ШІ (наприклад, генерація виправленого коду на основі аналізу існуючих помилок, використання *GAN* (*Generative Adversarial Networks*) або *RL* (*Reinforcement Learning*) для оптимізації структури коду).

Перевагами застосування нейронних мереж для аналізу програмного коду є*:*

– здатність апарату нейронних мереж до аналізу величезних обсягів даних і виявлення складних, неочевидних залежностей між різними частинами коду. Особливо це корисно для виявлення *code smells*, які можуть бути результатом комбінації кількох факторів;

– нейронні мережі можуть одночасно аналізувати велику кількість рядків коду, що робить їх ефективними для великих проєктів чи систем, де людський аналіз займе надто багато часу;

– нейронні мережі можуть бути навчені на значних наборах даних, які містять приклади "гарного" та "поганого" коду. Це дозволяє їм виявляти специфічні для певного проєкту чи команди *code smells*;

– нейронні мережі можуть враховувати контекст коду, такий як стиль програмування, доменна логіка або специфічні правила проєкту. Це робить їх гнучкішими порівняно з жорсткими правилами статичного аналізу;

– сучасні нейронні моделі можуть інтегруватися в інструменти розробки (*IDE*), надаючи розробникам моментальний зворотній зв’язок щодо потенційних проблем у коді;

– за допомогою механізмів онлайн-навчання нейронні мережі можуть покращувати свою продуктивність, адаптуючись до нових патернів або змін у проєкті.

Недоліками виступають: потреба значних обчислювальних потужностей для навчання та використання нейронних мереж, особливо для аналізу великих баз коду; рішення нейронних мереж можуть бути важкими для інтерпретації, оскільки вони часто працюють як “чорна скринька”; нейронні мережі можуть страждати від перенавчання, коли вони дуже точно вивчають тренувальні дані та починають давати неточні прогнози на нових даних. Це може призвести до помилкових спрацьовувань чи пропусків реальних проблем; якість роботи нейронної мережі залежить від якості тренувального набору даних. Якщо тренувальні дані містять помилки або не повністю відображають реальні ситуації, то модель може давати некоректні результати.

2. *Глибоке навчання для автоматичної генерації рефакторингових пропозицій*.

Глибоке навчання відкриває можливості для автоматизованого покращення якості коду, зокрема через:

– використання трансформерних моделей (*GPT*, *CodeBERT*, *Codex*) для прогнозування покращень у коді на основі виявлених недоліків;

– автоматичне формування рекомендацій щодо рефакторингу, наприклад, пропозиції замінити складні або дубльовані конструкції більш ефективними;

– навчання на великих наборах кодових баз для знаходження оптимальних рефакторингових рішень з урахуванням контексту та стилю кодування;

– поєднання технік *Natural Language Processing* (*NLP*) та *Computer Vision* для розуміння семантики та структури коду, що дозволяє виявляти складні зв’язки між компонентами програми.

Переваги: ефективне виявлення проблем у програмному коді (автоматична рефакторизація) і, відповідно, зменшення ручних операцій його аналізу; адаптація до різних мов програмування та платформ розробки, врахування контексту програмного коду (наприклад, стиль програмування, доменна логіка або специфічні правила проєкту); можливість навчатися на великих базах даних; глибокі моделі можуть постійно вчитися на нових даних, що дозволяє їм покращувати свою продуктивність і точність у міру отримання додаткових даних.

Недоліки: в деяких випадках ускладнене розуміння того, як саме модель прийшла до конкретної рекомендації рефакторингу; імовірність появи помилкових рекомендацій, особливо якщо тренувальні дані містять помилки або не повністю відображають реальні ситуації; потреба значних обчислювальних ресурсів; обмеження у розумінні семантики коду; потреба у спеціалізованих (експертних) знаннях із машинного навчання та обробки даних; обмеження у генерації оригінальних рішень.

3. *Алгоритми природної мови для аналізу коментарів і документації з метою покращення читабельності коду*.

Алгоритми обробки природної мови (*NLP*) можуть значно покращити читабельність і підтримку коду, аналізуючи коментарі та документацію.

Зазначені алгоритми здатні:

– автоматично оцінити якість коментарів програмного коду (а саме: виявити застарілі, суперечливі або нечіткі коментарі, оцінити відповідності коментарів до коду (наприклад,

наскільки вони описують актуальну функціональність), класифікувати текст для виявлення поганої документації);

– автоматично згенерувати коментарі та документацію до програмного коду. Тут може бути застосовано використання трансформерних моделей (*BERT*, *CodeBERT*, *GPT*) для генерації коментарів на основі коду, створення коротких пояснень до функцій, класів і методів, що спрощує роботу з великими кодовими базами, рефакторинг документації, щоб вона була більш структурованою та лаконічною;

– виявити невідповідності між кодом та документацією. Це забезпечується аналізом семантики коду та описів для знаходження розбіжностей, визначення змін у коді, що потребують оновлення документації;

– покращити стиль написання документації шляхом автоматичного виправлення граматичних і стилістичних помилок у коментарях, переформулювання надто складних або розпливчастих пояснень у більш зрозумілі.

Недоліки: алгоритми *NLP* часто мають труднощі з розумінням глибинного контексту та семантичної значимості тексту. Вони можуть неправильно інтерпретувати спеціалізовані терміни, абревіатури або доменну логіку, можуть не виявити суперечливість між описом у коментарях і реальним функціоналом коду; моделі *NLP* не зможуть покращити читабельність коду, якщо вхідні дані самі собою є некоректними; якщо модель навчена на обмеженому наборі даних, вона може генерувати рекомендації, які добре працюють лише для певного типу текстів. Це може призвести до некоректних пропозицій у нових або нетипових ситуаціях.

Отже, для ефективного використання нейронних мереж у цій задачі важливо:

– правильно налаштовувати моделі;

– забезпечити якість тренувальних даних;

– інтегрувати їх у процеси розробки так, щоб розробники могли довіряти рекомендаціям моделі.

Використання нейронних мереж значно покращує можливості статичного та динамічного аналізу, автоматизує процес виявлення проблем та сприяє ефективному рефакторингу коду.

Для успішного впровадження глибокого навчання у рефакторинг коду важливо:

– використовувати якісні тренувальні дані;

– забезпечувати прозорість і пояснення рекомендацій;

– інтегрувати систему у процеси розробки так, щоб розробники могли довіряти її пропозиціям.

Глибоке навчання має великий потенціал для автоматичної генерації рефакторингових пропозицій, оскільки воно може автоматизувати процес виявлення проблем і запропонувати покращення, які враховують контекст коду.

Для успішного застосування алгоритмів *NLP* важливо враховувати контекст, якість вхідних даних і специфіку проєкту. Ключовий фактор успіху – комбінація автоматичного аналізу з людським контролем, що забезпечує максимальну ефективність і точність рекомендацій.

Побудова моделі. Пропонується застосовувати рефакторинг програмного коду на основі ШІ із застосуванням механізмів статичного та динамічного аналізу не тільки для завдання виявлення вразливостей, помилок програмування, порушень стилю кодування, недоліків архітектури, але і для автоматизованого покращення коду, отримання безпечного та оптимізованого коду на основі виявлених проблем.

Функціональна модель рефакторингу об’єктно-орієнтованого коду на основі методів штучного інтелекту для забезпечення завдання безпеки програмного коду передбачає собою реалізацію наступних етапів (рис. 3):

1. *Збір даних за цільовий об’єктно-орієнтований програмний код як об’єкт аналізу*.

Цей функціональний модуль використовує вбудовані інструменти статичного та динамічного аналізу програмного об’єктно-орієнтованого коду. Інструменти статичного аналізу надають інформацію про помилки синтаксису, порушення стилю коду та потенційні вразливості.

Інструменти динамічного аналізу дають змогу виявити реальні загрози безпеці під час виконання коду.

Процеси статичного та динамічного аналізу програмного коду можуть виконуватися як паралельно, так і послідовно, залежно від контексту, мети аналізу та архітектури програмної системи.

Якщо процеси статичного і динамічного аналізів працюють паралельно, їхні результати потрібно об’єднати для подальшого використання (наприклад, для рефакторингу). У цьому випадку може знадобитися додаткова логіка для синхронізації та обробки цих даних. Це обмеження обумовило першочерговість у застосуванні інструментів статичного аналізу з передачею результатів як вхідних даних для динамічного аналізу.

Це дозволить отримати наступні переваги:

– динамічний аналіз може зосередитися лише на тих частинах коду, які були позначені як потенційно небезпечні під час статичного аналізу. Це економить час і ресурси;

– зменшується ймовірність “шуму” (помилкових сигналів) під час динамічного аналізу, оскільки він базується на надійних даних зі статичного аналізу;

– результати обох аналізів легше інтегрувати, оскільки вони мають чітку послідовність;

– можна сфокусуватися на найкритичніших областях програмного коду.

На цьому етапі формується набір даних, що містить інформацію про потенційні проблеми та вразливості.

2. *Навчання моделей ШІ*.

На даному етапі використовуються методи машинного навчання (*ML*) та глибинного навчання (*DL*) для створення моделей, що прогнозують наявність вразливостей.

У межах підготовчої фази цього етапу здійснюється попередня обробка даних, що отримані на попередньому етапі. Також передбачається створення набору ознак (*features*), таких як частота використання певних операторів, наявність небезпечних функцій тощо.

Виконавча фаза етапу навчання моделей використовує алгоритми *ML* (наприклад, *Random Forest*, *SVM*) для класифікації коду.

Застосування нейронних мереж (наприклад, рекурентних нейронних мереж *RNN* або трансформерів *Transformer*) передбачає виявлення складних шаблонів, які призводять до помилок або вразливостей.

Результатом цього етапу є навчені моделі, які можуть передбачати наявність проблем у коді.

3. *Автоматизований рефакторинг*.

Цей етап передбачає автоматичне виправлення виявлених проблем і також наявність двох фаз реалізації.

Спочатку здійснюється аналіз прогнозів, що передбачає використання навчених моделей для ідентифікації конкретних ділянок програмного коду, які потребують рефакторингу.

Далі здійснюється генерація безпечного коду:

– алгоритми ШІ (наприклад, генеративні моделі, такі як *GPT* або *CodeBERT*) створюють оптимізований та безпечний код;

– вносяться зміни у структуру класів, методів та інтерфейсів для покращення безпеки та продуктивності.

Результатом цього етапу є рефакторний код, який потенційно вважається безпечним і оптимізованим.

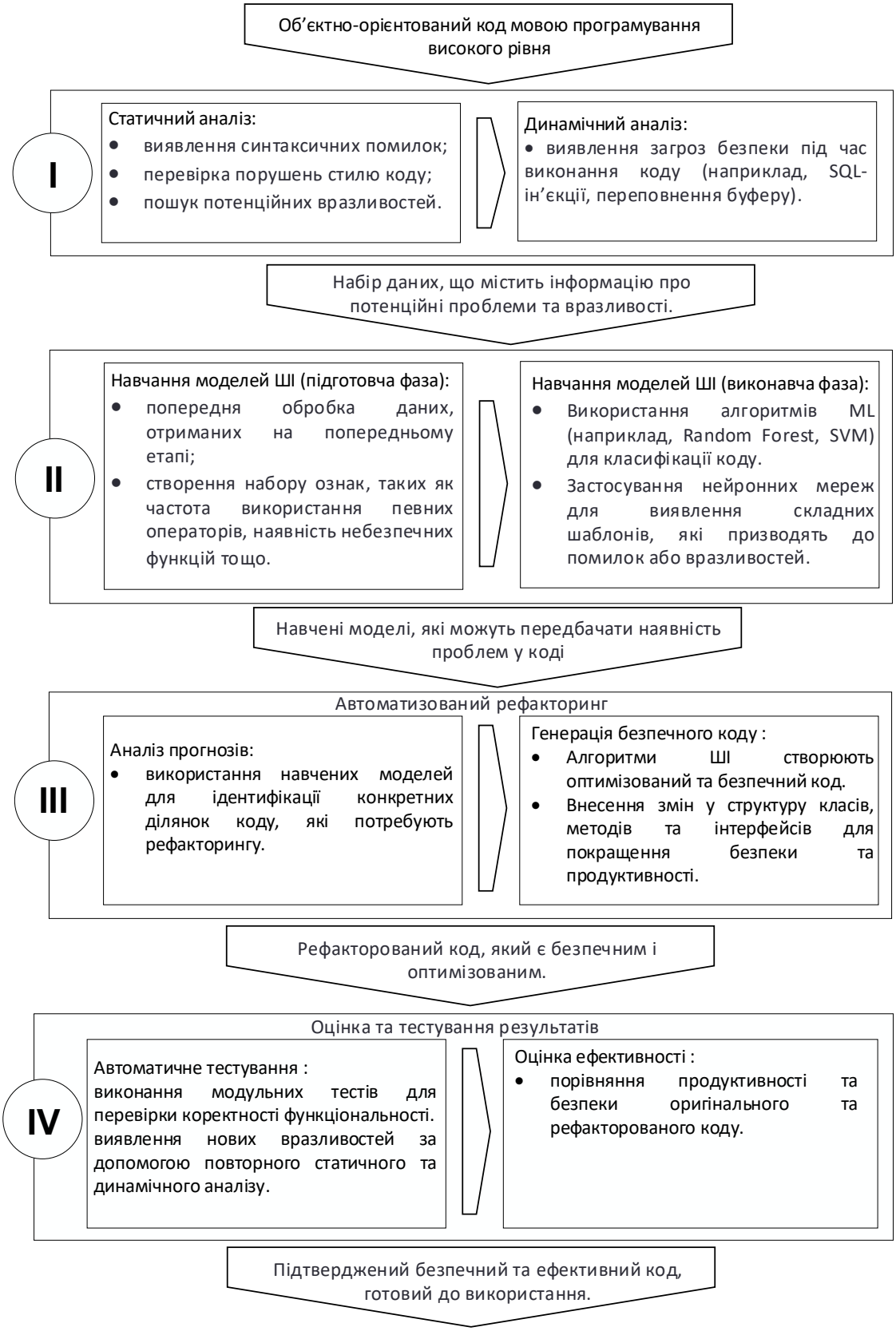

Рис. 3. Функціональна модель рефакторингу об'єктно-орієнтованого коду на основі методів штучного інтелекту

4. *Оцінка та тестування результатів.*

Останній етап функціональної моделі передбачає перевірку якості рефакторного коду.

Автоматичне тестування буде спрямоване на виконання модульних тестів для перевірки коректності функціональності.

Поява нових вразливостей може ініціалізувати собою повторну роботу інструментів статичного та динамічного аналізу.

Оцінка ефективності укладає собою завдання у порівнянні продуктивності та безпеки оригінального та рефакторного коду.

Табличне представлення функціональної моделі рефакторингу об'єктно-орієнтованого коду на основі методів штучного інтелекту за специфікацією *IDEF*0 (*Integration Definition for Function Modeling*) представлено у таблиці 1.

Таблиця 1

Модель рефакторингу об'єктно-орієнтованого коду на основі методів штучного інтелекту за специфікацією *IDEF*0

Процес	**КОНТРОЛЬ (*CONTROL*)**	**ВХІДНІ ДАНІ (*INPUTS*)**	**ВИХІДНІ ДАНІ (*OUTPUTS*)**	**МЕХАНІЗМ (*MECHANISM*)**
I	Процес аналізу коду	Набір даних, що містить інформацію про потенційні проблеми та ризики	Результати аналізу (статичні та динамічні помилки, порушення стилю коду тощо)	Алгоритми статичного та динамічного аналізу
II	Процес навчання моделей ML	Попередньо оброблені дані з результатами аналізу коду	Навчені моделі, які можуть передбачати наявність проблем у коді	Алгоритми машинного навчання (*Random Forest*, *SVM*, нейронні мережі тощо)
III	Процес автоматичного рефакторингу	Навчені моделі та аналізовані дані коду	Рефакторований код, який є безпечним і оптимізованим	Алгоритми генерації безпечного коду та оптимізації
IV	Процес оцінки ефективності та тестування	Рефакторований код	Підтверджений безпечний та ефективний код, готовий до використання	Модулі для тестування та оцінки продуктивності

За результатами етапів функціональної моделі рефакторингу об'єктно-орієнтованого коду на основі методів штучного інтелекту отримується підтверджений безпечний та ефективний код, готовий до використання за призначенням.

Таким чином, використання ШІ у рефакторингу об'єктно-орієнтованого коду дозволяє автоматизувати процеси покращення якості та безпеки, що робить програмне забезпечення більш захищеним та ефективним.

Висновки. Для забезпечення високих стандартів безпеки та якості в завданнях проєктування, розробки і супроводження інформаційних систем спеціального призначення важливо інтегрувати механізми їхньої реалізації у всі етапи життєвого циклу програмного забезпечення. Найбільш перспективним напрямом у вирішенні цього завдання виступає застосування процесу рефакторингу програмного коду. Цей процес постає превентивною мірою, яка запобігає потенційним атакам та зменшує ризики.

Рефакторинг програмного коду має чисельні обмеження, подолання яких покладається на долучення методів штучного інтелекту, особливо в завданнях статичного та динамічного аналізу, виявлення вразливостей, помилок програмування, порушень стилю кодування,

недоліків архітектури, автоматизованого покращення коду, отримання безпечного та оптимізованого коду на основі виявлених проблем.

Запропонована функціональна модель рефакторингу об’єктно-орієнтованого програмного коду на основі методів ШІ дозволяє автоматизувати процес покращення рівня безпеки та якості коду. Результати досліджень демонструють ефективність використання ШІ для виявлення вразливостей та автоматичного покращення структури коду.

Подальші дослідження можуть бути спрямовані на розширення можливостей моделі та її інтеграцію із сучасними інструментами розробки.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Goodfellow I., Bengio Y., Courville A. Deep Learning. MIT Press, 2016.
2. OWASP Foundation. OWASP Top Ten Security Risks. URL: https://owasp.org/www-project-top-ten/.
3. McConnell S. Code Complete. Microsoft Press, 2004.
4. Martin R. Clean Code: A Handbook of Agile Software Craftsmanship. Prentice Hall, 2008.
5. Fowler M. Refactoring: Improving the Design of Existing Code. Addison-Wesley, 2018.
6. Vasilescu B., Filkov V., Serebrenik A. Perceptions of Diversity on GitHub: A User Survey. Proceedings of CHI 2015.
7. Fields Jay, Harvie Shane, Fowler Martin. Refactoring Ruby Edition. Addison-Wesley, 2009
8. William C. Wake. Refactoring Workbook. Addison-Wesley, 2003. ISBN: 0321109295.
9. Feathers Michael. Working Effectively with Legacy Code. Prentice Hall, 2004.
10. Kerievsky Joshua. Refactoring to Patterns. Addison-Wesley, 2004.
11. Wagner Effective C#:5 Specific Ways to Improve Your C#/ 2015, 224 p.
12. Albahari Joseph, Albahari Ben. C# 7.0 in a Nutshell: The Definitive Reference, 2018. 1070 p.
13. Head First Design Patterns: A Brain-Friendly Guide: Building Extensible and Maintainable Object-Oriented Software Paperback – 5 Jan. 2021.
14. Refactoring: Improving the Design of Existing Code (Addison-Wesley Signature Series (Fowler)) Hardcover – Illustrated, 2 Jan. 2019.