УДК 004.05
DOI https://doi.org/10.32782/tnv-tech.2025.2.17

МЕТОДИ ПІДВИЩЕННЯ КОРЕКТНОСТІ ТА БЕЗПЕЧНОСТІ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ: СУЧАСНИЙ ФУНКЦІОНАЛ КОМПІЛЯТОРІВ ТА ІНШІ ПІДХОДИ

Рихальський О. Ю. *– викладач кафедри комп'ютерних наук та програмної інженерії ПВНЗ «Європейський університет»*
ORCID ID: 0009-0000-6892-9420

У контексті стрімкого впровадження цифрових технологій програмне забезпечення відіграє визначальну роль у функціонуванні критично важливих сфер – від фінансових і медичних систем до інфраструктурних рішень, засобів безпеки та промислової автоматизації. Відповідно, зростають вимоги до якості, надійності та безпечності програмних систем. Одним із ключових чинників, що формують ці характеристики, є коректність програмного забезпечення, тобто відповідність його поведінки формальним або неформальним специфікаціям в усіх допустимих ситуаціях.

Забезпечення коректності є складним завданням, що потребує як глибокого розуміння предметної області, так і ретельного технічного моделювання. Типовими причинами відхилень є логічні, структурні чи концептуальні помилки, що можуть спричинити дестабілізацію роботи системи, втрату або пошкодження даних, а також створення потенційних вразливостей безпеки.

З огляду на це, особливого значення набувають інструменти раннього виявлення та попередження дефектів. Серед них вирізняються сучасні компілятори, функціонал яких уже виходить за межі синтаксичного аналізу. Вони здатні виконувати глибоку семантичну перевірку – зокрема, щодо повноти обробки умов, допустимості значень параметрів, коректності індексації тощо. Важливу роль відіграють і засоби статичного аналізу коду, що дозволяють оцінювати поведінку програм без їх фактичного виконання.

Попри наявність потужних інструментів, у сучасній інженерній практиці часто домінує орієнтація на швидке виведення продукту на ринок. У результаті питання коректності частково ігноруються – особливо в умовах стартап-культури або гнучких методологій. Водночас у високоризикових сферах, таких як охорона здоров'я, енергетика чи безпека, навіть незначні відхилення можуть мати фатальні наслідки.

У статті представлено аналіз низки класичних і сучасних публікацій, присвячених підвищенню якості програмного забезпечення. Деякі з них пропонують систематизовані методи забезпечення коректності, інші – концептуальні підходи до створення програм з мінімальними можливостями помилок. Особливу увагу приділено засобам контролю коректності на рівні компіляції.

Ключові слова: *інформаційні технології, моделювання даних, коректність програмного забезпечення, забезпечення якості, надійність програмних систем, параметри, складні системи, часткові функції, повнота варіантів, обробка граничних випадків.*

Rykhalskyi O. Yu. Methods for improving software correctness and security: modern compiler functionality and other approaches

In the context of the rapid adoption of digital technologies, software plays a crucial role in the functioning of critically important domains – ranging from financial and medical systems to infrastructure solutions, security tools, and industrial automation. Accordingly, the demands on the quality, reliability, and security of software systems continue to grow. One of the key factors shaping these characteristics is software correctness, defined as the compliance of a program's behavior with its formal or informal specification in all permissible situations.

Ensuring correctness is a complex task that requires both deep understanding of the application domain and careful technical modeling. Common sources of deviations include logical, structural, or conceptual errors, which may destabilize system operation, lead to data loss or corruption, and in some cases – introduce serious security vulnerabilities.

Given these risks, tools for early detection and prevention of defects become particularly significant. Among them, modern compilers stand out for their expanded functionality, which goes beyond traditional syntactic analysis. They are now capable of performing advanced

semantic checks – including completeness of condition handling, validity of parameter values, index correctness, and more. Static code analysis tools also play an essential role, allowing assessment of program behavior without actual execution.

Despite the availability of such powerful tools, current engineering practice often prioritizes rapid product deployment. As a result, correctness concerns are frequently sidelined – especially in startup environments or under agile methodologies. However, in high-risk sectors such as healthcare, energy, or security, even minor behavioral deviations can have catastrophic consequences.

This article presents an analysis of both classical and contemporary publications devoted to improving software quality. Some of them offer structured methods for ensuring correctness, while others propose conceptual approaches to designing software that minimizes the likelihood of errors. Particular attention is paid to mechanisms for correctness verification at the compilation stage.

Key words: *information technology, data modeling, software correctness, quality assurance, reliability of software systems, parameters, complex systems, partial functions, exhaustiveness checking, edge case handling.*

Постановка проблеми. У сучасній практиці розробки програмного забезпечення дедалі більше домінує орієнтація на швидкість створення та виведення продуктів на ринок. Такий підхід, хоча й виправданий в умовах динамічного бізнес-середовища, нерідко супроводжується зниженням уваги до глибокого аналізу предметної області та проектування програмної системи. Як наслідок, ключові концепти предметної галузі або залишаються неформалізованими, або втрачаються під час переходу від вимог до реалізації. Частина знань експертів-галузевиків зберігається лише в неявній формі (інколи у вигляді усних коментарів або припущень), що унеможливлює подальшу перевірку та систематизацію цих знань у програмному коді.

Додаткову складність становить той факт, що розробники, як правило, використовують універсальні або так звані «стандартні» мови програмування, які були спроектованими без урахування специфіки формального контролю коректності. У результаті розробка ведеться за допомогою інструментів, які не мають вбудованих механізмів для вираження обмежень, перевірки повноти варіантів або явної специфікації допустимих значень.

Ще однією поширеною практикою є використання надто загальних типів (наприклад, String, Int, Any), які не фіксують семантику значень і не дозволяють компілятору виявити помилки, пов’язані з некоректним використанням даних. Це створює додаткове навантаження на програмістів та знижує здатність інструментальних засобів до виявлення потенційних дефектів на ранніх етапах.

Крім того, в багатьох програмних системах спостерігається схильність до реалізації лише так званого «щасливого сценарію» (happy path), без належної обробки граничних, виняткових або помилкових ситуацій. Такий підхід істотно знижує надійність програмних рішень, особливо в критичних застосуваннях, де неочікувана поведінка може мати серйозні наслідки.

Таким чином, існує нагальна потреба в переосмисленні та структуруванні підходів до забезпечення коректності програмного забезпечення – з урахуванням сучасної практики, необхідності формалізації предметних знань і використання інструментів, що підтримують виявлення помилок на етапі проєктування та компіляції.

Аналіз останніх досліджень і публікацій. Аналіз джерел [1–15] дозволив здійснити попередню класифікацію існуючих підходів до підвищення якості та коректності програмного забезпечення. Зазначені публікації репрезентують різні школи, дослідницькі традиції та практичні контексти, у яких здійснюється розробка надійних програмних систем.

У працях [1, 2, 4, 10, 11] представлено низку так званих «правильних» рішень у вигляді найбільш поширених шаблонів проєктування. Ці підходи базуються на накопиченому досвіді спільноти розробників та орієнтовані на повторне використання ефективних архітектурних рішень, що сприяє підвищенню стабільності програмних систем.

У роботах [5, 7, 12, 15] здійснено ретроспективний аналіз практики розробки програмного забезпечення за останні два десятиліття. Автори ідентифікують ключові помилки та прорахунки, які призводили до зниження якості, а також формують систематизований перелік шаблонів і антипатернів, що дозволяє розмежувати конструктивні та деструктивні проєктні рішення.

Дослідження [3, 9, 13] акцентують увагу на важливості проєктування коректної програмної архітектури на ранніх етапах життєвого циклу. Особливу увагу приділено ідеї розділення опису даних та коду, який обробляє ці дані, що підвищує прозорість та передбачуваність системної поведінки.

У працях [3, 6, 12] розглядаються філософські та когнітивні аспекти, пов'язані з проєктуванням програмного забезпечення. Автори порушують питання системних причин повторюваності типових помилок навіть у середовищі висококваліфікованих спеціалістів, підкреслюючи важливість дисциплінованого мислення та постійного аналізу помилок у проєктах.

У [6, 12] представлено дослідження залежності між категорією інженера та ймовірністю припущення помилки. Зокрема, аналізується, чому деякі групи розробників демонструють істотно вищу якість коду, ідентифікуючи фактори, які сприяють зменшенню кількості помилок у 5–10 разів порівняно з іншими категоріями фахівців.

Особливе місце посідають роботи [13, 14], у яких запропоновано формалізований підхід до моделювання даних, що дозволяє на рівні типів унеможливити перехід системи в некоректний стан. Методологія ґрунтується на використанні систем типів із жорсткою, але водночас гнучкою структурою, що дає змогу точно та виразно описувати семантику даних і допустимих операцій над ними.

Водночас слід зазначити, що кожна з розглянутих публікацій має власний предметний фокус, методологію та межі застосовності. Це обумовлює необхідність комплексного підходу до інтеграції різних підходів у межах єдиної системи забезпечення коректності програмного забезпечення.

Метою статті є формалізація та систематизація існуючих підходів до забезпечення коректності програмного забезпечення, представлених у сучасній науковій та інженерній літературі. Автор прагне узагальнити ці підходи, виявити їх спільні структурні риси, спростити складні концепції для практичного застосування та надати інтерпретацію на рівні абстракцій вищих категорій.

Методи досліджень. У цьому дослідженні застосовано системний підхід, що передбачає всебічний аналіз чинників, які впливають на коректність і безпечність програмного забезпечення. Основою методології виступає систематизація наявних теоретичних і практичних підходів, виявлених у сучасній літературі [14, 15], а також співставлення цих підходів з типовими сценаріями сучасної розробки програмних систем [1, 7].

Аналіз джерел здійснювався із урахуванням таких критеріїв, як: наявність або відсутність формалізації коректності; тип інструментів (компілятори, системи типів, засоби верифікації); рівень абстракції, на якому відбувається контроль помилок (код, архітектура, предметна модель); відповідність рішень сучасним практикам програмної інженерії.

Окрему увагу приділено виявленню загальних принципів, які лежать в основі різних підходів до гарантування правильності поведінки програм: від безпечного моделювання на рівні типів до розширеного семантичного аналізу в компіляторах. На основі цих принципів сформовано попередню класифікацію інструментів та методів, що сприяють підвищенню надійності коду.

Важливою складовою дослідження є власний практичний досвід автора, який охоплює понад 30 років активної роботи в галузі програмування та понад 10 років викладання дисциплін, пов'язаних із розробкою програмного забезпечення. Цей досвід дозволив поєднати аналітичну глибину з практичною релевантністю та критично оцінити ефективність і доцільність окремих підходів з точки зору їх реального застосування.

Завдяки поєднанню теоретичного аналізу, практичного осмислення та міждисциплінарного контексту, дослідження дозволяє сформулювати низку узагальнених висновків і окреслити перспективні напрями подальшого розвитку методів забезпечення коректності програмного забезпечення.

Програмування – це просто. На перший погляд, програмування виглядає відносно просто. У своїй основі комп'ютер виконує дуже обмежений набір елементарних операцій, які можна згрупувати у три фундаментальні категорії.

По-перше, це арифметичні операції, серед яких найважливішою є додавання. У більшості архітектур, апаратне забезпечення здатне лише додавати, а всі інші дії – наприклад, віднімання, чи множення – зводяться до додавання з певними перетвореннями.

По-друге, комп'ютер вміє керувати потоком виконання. Це здійснюється через переходи – безумовні або умовні. Умовні переходи спираються на результати попередніх операцій, які впливають на спеціальні регістри або прапорці. Наприклад, перевірка, чи дорівнює значення нулю, чи є воно від'ємним або чи сталася арифметична помилка, дає змогу приймати рішення щодо подальшого ходу обчислень.

По-третє, система забезпечує доступ до пам'яті. Це дві взаємо-зворотні дії: зчитування з певної адреси та запис у неї. Усе, що ми сприймаємо як змінні, масиви, об'єкти чи структури, є лише різними способами організації доступу до ділянок пам'яті.

Таким чином, функціональні можливості сучасних мов програмування можуть бути зведені до трьох базових категорій інструкцій: обчислювальні операції, керування порядком виконання та роботою з пам'яттю. Саме ця структурна простота обґрунтовує твердження, що будь-який алгоритм – незалежно від його складності – принципово може бути реалізований шляхом поєднання цих елементарних конструкцій машинної логіки.

Програмування – це складно. Однак на практиці програмування є значно складнішим, ніж цей теоретичний мінімум [3]. Складність стрімко зростає разом з масштабом програмного продукту [1]. Варто лише вийти за межі кількох сотень рядків коду – і виникає проблема втрати контексту. Інженер вже не може одночасно тримати в пам'яті всю взаємодію між частинами системи [10]. Кожен новий модуль, функція або залежність збільшує кількість можливих комбінацій, що призводить до експоненційного зростання складності підтримки, тестування і модифікації коду [9].

У таких умовах навіть просте на перший погляд завдання – реалізувати нову функцію – перетворюється на низку дрібних рішень, залежних від вже наявної архітектури, зв'язків між модулями, обмежень сторонніх бібліотек, стану даних, очікуваної поведінки системи та прихованих припущень. Усе це призводить до того, що прогнозування термінів реалізації нової функціональності стає дедалі

менш точним. Глибина залежностей та потенційні ризики залишаються непомітними до моменту безпосередньої реалізації [3].

Таким чином, програмування поєднує в собі граничну простоту на рівні базових операцій і величезну складність на рівні реальних систем, що робить цю діяльність унікальною за своєю природою: формально визначеною, але водночас творчою, непередбачуваною і схильною до помилок.

Унаслідок стрімкого зростання складності програмне забезпечення дедалі частіше втрачає ознаки надійності [4]. Системи, які складаються з тисяч і мільйонів рядків коду, нерідко містять приховані помилки, що не виявляються під час звичайного тестування. Це призводить до часткової некоректності роботи: непередбачуваних відмов, неконсистентних станів даних або неправильних реакцій на зовнішні події. У критичних випадках програмне забезпечення може стати джерелом небезпеки – як для бізнес-процесів, так і для людей.

Коректність як базова умова надійності та безпечності програмного забезпечення. Коректність є базовою характеристикою якості програмного забезпечення, що визначає його здатність виконувати поставлені задачі, забезпечувати цілісність даних і уникати небезпечної поведінки. Вона розглядається як відповідність поведінки програми її специфікації – формальному [14] або неформальному опису очікуваного результату для всіх допустимих вхідних даних. У рамках формальних методів коректність є предметом верифікації, яка передбачає доведення того, що програма задовольняє визначені інваріанти. Це охоплює як правильність обчислень, так і логічну узгодженість усіх гілок виконання.

Тісно пов'язані з коректністю поняття надійності та безпечності. Надійність описує здатність системи зберігати правильну поведінку в часі та під впливом змін середовища. Безпечність – здатність запобігати несанкціонованим діям і гарантувати відповідність поведінки встановленим правилам доступу.

Обидві характеристики випливають із коректності: некоректно реалізована система апріорі не може бути ані надійною, ані безпечною. Саме тому коректність є теоретичною основою, на якій вибудовуються стратегії забезпечення якості. Практичні засоби досягнення коректності включають формальні методи, статичний аналіз, системи типів, модульне тестування та інші підходи. Їх ефективне застосування потребує чіткого уявлення про те, що таке «коректна поведінка» та як вона формалізується у рамках обраної мови чи платформи.

Розглянемо типові помилки та визначимо напрямки вирішення або запобігання.

1. Код який описує дані є зв'язаний з кодом який працює з даними. Мабуть найпоширеніша проблема, яка пов'язана з тим, що код потрібно писати швидко, а потім, коли код вже працює, майже ніколи не вистачає часу, а інколи й експертизи розділити код і зробити правильно.

```
class User(id: Int, name: String) {
def save(f: File) = ???
def load(f: File) = ???
}
```

Як результат, коли ми змінюємо або реалізацію коду, який працює з даними, або код, який описує дані – кількість змін є дуже великою і стає не достатньо зрозуміло, що відбувається. Це призводить до величезних витрат часу за ризиків під час внесення змін.

Рішення, яке пропонується в [4, 9] – розділити опис даних та роботу з даними на два різних файли, можливо навіть в різних пакетах. Відповідно зміни можна вносити по черзі, тим самим зменшуючи ризики. Код буде виглядати наступним чином:

```
class User(id: Int, name: String)

class UserOps {
def save(f: File) = ???
def load(f: File) = ???
}
```

2. Проблема частковості функцій в контексті програмної інженерії. Однією з фундаментальних тем у проєктуванні програмного забезпечення є категорія так званих неповних функцій [9, 13, 14]. Така функція визначається як часткова функція – тобто функція, яка не визначена на всій області значень її вхідних параметрів. На відміну від повної функції, яка гарантує коректне та очікуване значення для будь-якого допустимого аргументу згідно з її типом, часткова функція може спричинити помилку під час виконання, або навіть призвести до аварійного завершення програми.

Розглянемо приклад часткової функції, яка має на меті конвертацію рядка у ціле число:

```
def convert(s: String): Int = ???
```

На перший погляд, функція виглядає абсолютно коректною: вона приймає рядок та повертає відповідне ціле число. Проте, якщо передати у неї рядок, що не є числом, наприклад «123a», то поведінка буде залежати від мови програмування або її стандартної бібліотеки. У деяких мовах буде згенеровано виключення (exception), в інших – повернуто спеціальне значення (null, undefined). У будь-якому з варіантів, функція convert не виконує свою обіцянку, закладену у її сигнатурі, яка передбачає повну визначеність на множині всіх можливих рядків.

Таким чином, перед нами постає важливе питання: на кому лежить відповідальність за перевірку допустимості вхідних даних? Чи повинен розробник, який викликає цю функцію, заздалегідь здійснити перевірку рядка на валідність? Чи все ж таки відповідальність має бути на стороні автора функції, який повинен був закласти в сигнатуру або реалізацію механізм обробки винятків чи інші засоби безпечного виконання?

Схожі приклади можна навести і для інших функцій:

```
def inverse(x: Double): Double = 1 / x
```

Функція обчислює обернене значення до x, однак математично вона не визначена для нуля. Тим не менш, її сигнатура $Double \Rightarrow Double$ не несе жодної інформації про те, що вхідне значення не повинно дорівнювати нулю. Розробник, який використовує цю функцію, може не знати або забути про це обмеження, що призведе до виникнення помилки виконання або генерації нечислового значення (Infinity, NaN, null) або exception, залежно від конкретної мови чи платформи.

Ще один приклад:

```
def sqrt(x: Double): Double = ???
```

Корінь квадратний, як відомо з елементарної математики, не визначений для від'ємних чисел у множині дійсних чисел. Проте і тут сигнатура $Double \Rightarrow Double$

не передає цього обмеження. Розробник змушений або самостійно перед викликом перевіряти вхідні дані, або покладатися на непередбачувану поведінку під час виконання.

Проблема неповноти функцій не обмежується лише арифметичними чи математичними прикладами. Вона широко поширена в роботі з системними ресурсами, як-от файловими системами. Розглянемо функцію:.

```
def readFile(filename: String): String = ???
```

Навіть не знаючи реалізації, можна спрогнозувати потенційні помилки: файл може не існувати, до нього можуть бути відсутні права доступу, диск може бути недоступним тощо. Але всі ці сценарії залишаються за межами сигнатури функції *String* ⇒ *String*, що знову вводить в оману потенційного користувача коду.

Ці приклади демонструють одну з найсуттєвіших проблем: сигнатура функції не відображає повною мірою її поведінку, особливо у разі виникнення помилок. Звідси виникає питання – яким чином забезпечити більш точне, інформативне й безпечне програмування?

Як ми бачимо, частковість функцій є глибокою концептуальною проблемою, яка часто залишається непоміченою, особливо у мовах із слабко вираженими системами типів. Визнання частковості як особливої властивості функцій, що потребує відповідної уваги на рівні сигнатур, дозволить будувати більш надійне програмне забезпечення, де поведінка кожного компонента чітко визначена й контрольована.

Існує два основних вирішення цієї проблеми. Одне рішення, детально розглядається в [9, 11, 14] – це використання алгебраїчних типів даних для репрезентації факту того, що функція часткова. Після змін, які пропонуються, код буде виглядати наступним чином:

```
def convert(s: String): Option[Int] = ???

def inverse(x: Double): Option[Double] = ???

def f1(x: Double): Option[Double] = ???

def readFile(filename: String): Either[Exception, String] = ???
```

Використання алгебраїчного типу даних Option підкреслює факт можливості «не отримання» результату і гарантує той факт, що обидва варіанти *Some*(*x*) та None будуть опрацьовані. Використання алгебраїчного типу даних Either підкреслює факт можливості двох можливих варіантів обчислень: *Left*(*e*) (помилка) та *Right*(*a*) (результат), і також гарантує той факт, що обидва варіанти *Right*(*a*) та *Left*(*e*) будуть опрацьовані.

Наступне рішення, які пропонуються в [8, 13] це використання бібліотеки Refined, яке дає можливість перевіряти вхідні параметри ще на етапі компіляції, і в цьому випадку сигнатури методів будуть виглядати іншим чином:

```
def inverse(x: Double Refined NonZero): Double = ???

def sqrt(x: Double Refined NonNegative): Double = ???
```

В даному випадку при виклику функцій

```
val y = inverse(0)
```

або

```
val y = sqrt(-9)
```

цей код навіть не буде скомпільованим.

Треба звернути увагу на той факт, що ці два підходи не замінюють один одного – а доповнюють, тому що не завжди можливо перевірити вхідний параметр. Можливо файл і існує, але в момент запуску вже не буде існувати або не буде прав на доступ до нього.

Але принципово, ці два підходи однакові, фони формують повну функцію, яка гарантує поведінку і відповідний результат при будь якому вхідному значенні.

3. Обробка не всіх можливих варіантів як джерело помилок. Однією з найбільш поширених і водночас критичних помилок у програмній інженерії є неповна обробка можливих варіантів [1, 2, 10]. У мовах програмування ця проблема часто проявляється через відсутність блоку else у конструкціях if-then, а також у пропущених варіантах при використанні операторів switch (у Java) або match (у Scala). Наслідком таких упущень стає неочікувана поведінка програми, яка може призвести до логічних помилок, порушення узгодженості станів системи, втрати даних, або аварійного завершення виконання.

3.1. Відсутність else у конструкції if-then.

У мовах, де if не є виразом, а інструкцією (як у Java), розробник може легко проігнорувати або забути негативну гілку умовного переходу:

```
void checkAccess(boolean isAdmin) {
if (isAdmin) {
System.out.println(“Access granted”);
}
// else-гілка відсутня
}
```

У цьому фрагменті поведінка функції у разі isAdmin == false не визначена явно. Така відсутність else не призводить до синтаксичної помилки, але логіка залишається неповною. Якщо програма працює у контекстах, де деталі доступу є критичними (наприклад банківська система, медицина), таке «мовчання» може стати серйозною вразливістю. Повна форма:

```
if (isAdmin) {
System.out.println(“Access granted”);
} else {
System.out.println(“Access denied”);
}
```

У мові програмування Scala, де if – це вираз, такий підхід може спричинити ще складніші помилки, оскільки if без else повертає Any, а не значення очікуваного типу:

```
val result: String = if (isAdmin) “Access granted” // else
відсутній
```

В цьому випадку компілятор зупиниться з помилкою, тому що результуючий тип Any не є типом String.

Але у випадку

```
val result = if (isAdmin) “Access granted” // else відсутній
```

result отримає тип Any, що буде зрозуміло за декілька рядків або сторінок коду, де цей result буде використовуватися. Це створює додаткову неоднозначність і порушує структуру.

3.2. Неповний switch у Java з enum.

У Java при роботі з переліками (enum) існує поширена помилка: не всі можливі значення переліку обробляються в switch. Наприклад:

```
enum Status { OK, ERROR, TIMEOUT }

void handleStatus(Status status) {
switch (status) {
case OK -> System.out.println("All good");
case ERROR -> System.out.println("Something went wrong");
// TIMEOUT не оброблено!
}
}
```

Такий код буде компілюватися успішно, але якщо у виконанні буде передано Status.TIMEOUT, жоден з кейсів не спрацює, і switch нічого не зробить. Це залишає програму у неочікуваному або неузгодженому стані. Ще гірше, якщо після switch система вважає, що статус було оброблено й продовжує дії, засновані на хибному припущенні. Цей клас помилок вкрай важко знаходити.

Щоб уникнути цього, слід або явно обробити всі варіанти:

```
case TIMEOUT -> System.out.println("Request timed out");
```

Або додати захисний default:

```
default -> throw new IllegalStateException("Unexpected: " +
status);
```

В даному випадку гілка default «викине» виключення, і програміст буде знати, шо існує значення, яке залишилося необробленим. Але default має серйозну потенційну проблему – у випадку додавання елементів до enum, під гілку default можуть попадати декілька різних значень з enum, що в перспективі буде складно знайти та виправити.

3.3. Неповний pattern matching у Scala при використанні trait.

У Scala потужним інструментом для обмеження множини можливих типів є trait. Якщо в конструкції match всі підтипи цього trait не оброблені, це призводить до потенційно хибної логіки або виникнення exception на етапі виконання:

```
sealed trait Command
case object Start extends Command
case object Stop extends Command
case object Pause extends Command

def handle(cmd: Command): String = cmd match {
case Start => «Started»
case Stop => «Stopped»
// Pause не оброблено
}
```

Компілятор, в залежності від налаштувань, може навіть не видати попередження. Програма залишиться з «діркою» в логіці, і при передачі Pause буде викинуто MatchError у на етапі виконання.

Захищений варіант може виглядати так:

```
def handle(cmd: Command): String = cmd match {
case Start => “Started”
case Stop => “Stopped”
case Pause => “Paused”
}
```

В Scala також існує default, він має трохи інший синтаксис, але краще підкреслює семантику:

```
case _ => throw new IllegalArgumentException(“Unknown command”)
```

Семантика цього оператора – «все інше». Зазвичай використовується коли нас дійсно не цікавлять інші варіанти та використовується коли в trait є багато елементів, а цікавлять нас тільки декілька. Застереження ті самі, що і в Java, при додаванні елементів в trait дуже легко припуститися помилки і забути обробити додане значення.

3.4. Неповна обробка варіантів із Option або Either у Scala.

Ще одне поширене джерело помилок – це ігнорування одного з можливих варіантів алгебраїчного типу:

```
val result: Option[String] = ???

val message = result match {
case Some(value) => value
// None не оброблено!
}
```

У даному прикладі, якщо result буде None, це призведе до exception на етапі виконання. Такі помилки особливо небезпечні, оскільки Option використовується для репрезентації відсутності значення – тобто явно вказує, що None є очікуваним і допустимим варіантом. Його ігнорування – грубе порушення контракту типу.

Аналогічна ситуація з Either, де лише одна сторона оброблена:

```
val result: Either[String, Int] = ???

val number = result match {
case Right(n) => n
// Left не оброблено!
}
```

У випадку Right код спрацює нормально, у випадку Left – виникне exception на етапі виконання. Все виглядає цілком логічним, тому що в Scala алгебраїчні типи моделюються за допомогою sealed trait.

3.5. Логіка з припущенням про повноту переліку.

У Java частою помилкою є жорстке припущення, що перелік не буде розширено. Наприклад, маємо такий код:

```
enum Mode { MANUAL, AUTO }

String describe(Mode mode) {
return switch (mode) {
case MANUAL -> “User control”;
```

```
case AUTO -> “Automatic control”;
};
}
```

Цей код виглядає коректно, однак при додаванні нового значення SEMI_AUTO в enum, метод describe більше не є повним, і компілятор може не вказати на помилку – залежно від налаштувань. Це вказує на необхідність ретельного перегляду логіки при еволюції переліків. Необроблені варіанти – це мовчазні помилки, які не завжди проявляються під час компіляції, але призводять до фатальних наслідків у виконанні.

Рішення які пропонуються [9, 13, 14] це використання сучасних компіляторів Java 21, Scala, Rust, Haskell з відповідними налаштуваннями та використання концепції sealed, яка накладає обмеження, що всі під-типи будуть оголошені в тому самому файлі і компілятор буде зупинятися с помилкою «Not all variants covered». Це дозволить знаходити помилки такого класу ще на етапі компіляції, навіть до фази написання тестів.

4. Обробка виключень (exceptions). Обробка винятків є ключовим механізмом забезпечення надійності програмного забезпечення. В багатьох мовах програмування, зокрема Java, Scala, Python, Rust, вона використовується для виявлення та обробки непередбачених ситуацій, які виникають під час виконання програми. Найчастіше винятки виникають у випадках, коли функції є неповними – тобто не охоплюють усі можливі вхідні значення або граничні ситуації. Наприклад, спроба зчитування числа з нечислового рядка, доступ до елементу поза межами масиву або ділення на нуль, не знайдений файл – усе це типові джерела винятків. Причиною є те, що такі ситуації або не передбачені логікою функції, або ігноруються на етапі розробки. Умовна конструкція if без else, оператор match без обробки всіх варіантів – усе це створює потенційні вразливості. Тому ефективна обробка винятків передбачає не лише технічну реалізацію try-catch, але й попередження помилок через повні функції, перевірку вхідних даних та додатковий фокус на аналізі граничних випадків.

Ми бачимо що ця проблема йде майже впритул з частковими функціями. Тому основним рішенням є фокус на повноті функцій.

Дуже часто в коді з'являються конструкції які виглядають наступним чином:

```
try {
// код який може «зламатися»
} catch (Exception x) {
}
```

Зазвичай цей код з'являється у «не дуже» досвідчених інженерів, які не приділяють достатньо уваги причині того, що код може «зламатися» і чому саме. Цей тип помилки вкрай важко знайти, тому що помилка є «з'їденою» і жодним чином себе не проявляє, окрім інколи неправильно працюючої логіки. Гарним рішенням є використання алгебраїчного типу Either, який підкреслить можливість факту наявності помилки і відповідно помилка буде оброблена пізніше. Код буде виглядати наступним чином:

```
def doSomething(): Either[Exception, Int] =
try {
// код який може «зламатися»
val a = ???
```

```
Right(a)
} catch (Exception x) {
Left(x)
}
```

Фактично ми «зберігаємо» помилку і даємо можливість обробити її пізніше.

5. Обробка великих обсягів даних. Під час роботи з великими файлами, потоками даних або іншими джерелами, що генерують значні обсяги інформації, виникає серйозна проблема – споживання пам'яті. Якщо програмне забезпечення намагається завантажити всі дані одразу в оперативну пам'ять без відповідного контролю, це призводить до її переповнення та аварійного завершення роботи програми [13, 15]. Такі помилки є типовими прикладами недостатнього аналізу предметної площини, особливо якщо не передбачено захисту або обмежень на розмір вхідних даних.

Щоб запобігти подібним ситуаціям, у сучасних мовах програмування передбачено спеціальні підходи до потокової обробки. Наприклад, у мові Scala популярним є використання бібліотеки fs2, яка реалізує концепцію потоків з керованим споживанням пам'яті та підтримкою асинхронної обробки. У мові Rust подібну роль виконують ітератори, які дозволяють зчитувати та обробляти елементи по одному, завантажуючи нові лише після завершення роботи з попередніми, що гарантує ефективне використання ресурсів.

6. Відсутність перевірки вхідних даних як за типом, так і за обсягом. Однією з найбільш поширених і водночас критичних проблем, що впливають на коректність і безпечність програмного забезпечення, є відсутність належної перевірки вхідних даних. Йдеться як про відсутність перевірки типів і форматів, так і про нехтування обмеженнями на обсяг, структуру, кількість параметрів або розмір переданих повідомлень. У ситуаціях, коли система очікує добре структуровані дані певного типу, але натомість отримує некоректні або шкідливо сформовані вхідні значення, можуть виникнути як помилки виконання, так і суттєві загрози для інформаційної безпеки.

До типових проявів такої проблеми належать: переповнення буфера при обробці вхідних рядків, атаки типу SQL-ін'єкцій у випадках недостатньої фільтрації запитів до бази даних, відсутність обмежень на розмір файлів або довжину рядків, переданих через API, а також ситуації, коли користувач або зловмисник може передати надмірну кількість параметрів або запитів, що призводить до перевантаження системи.

Особливо небезпечною є практика повного зчитування вхідних даних у пам'ять без попереднього аналізу їх обсягу. У такому випадку навіть одиничний запит може спричинити вичерпання оперативної пам'яті, що веде до деградації продуктивності, аварійного завершення процесу або повної відмови сервісу.

Для зменшення ризиків, пов'язаних з неконтрольованими вхідними даними, пропонується низка практичних рішень [13, 15]. По-перше, це сувора перевірка типів, допустимих значень та структури на всіх рівнях обробки – від зовнішнього API до внутрішніх сервісів. По-друге, впровадження обмежень на розмір запитів, файлів або параметрів, з відповідним сповіщенням користувача. По-третє, використання потокової обробки, яка була розібрана вище. Крім того, розробка програм із врахуванням граничних випадків та навмисне тестування сценаріїв із перевищенням допустимих меж вхідних параметрів. Також важливо впроваджувати засоби спостереження за аномальною активн-істю, що дозволяє захиститися від перевантаження системи через масові запити.

7. Відсутність перевірки крайових ситуацій: системна вразливість до непередбачуваних помилок. Однією з поширених причин нестабільної роботи програмного забезпечення є відсутність перевірки крайових або граничних ситуацій, які не охоплюються під час розробки або тестування. До таких ситуацій належать обробка масивів нульової довжини, значень, що дорівнюють мінімальному або максимальному допустимому порогу, а також робота з порожніми файлами, їх відсутність або наявність у неправильному форматі. Часто подібні стани не розглядаються в логіці коду, що призводить до виникнення виключень, помилок виконання або повної зупинки програми.

Відсутність обробки крайових випадків робить систему крихкою та вразливою до змін в оточенні, непередбачуваних дій користувача або нетипових даних. Для підвищення надійності рекомендується як на етапі проєктування, так і під час тестування враховувати максимально широкий спектр сценаріїв, у тому числі виняткових. Системне мислення та практики захисного програмування дозволяють мінімізувати ризики та підвищити надійність програмного забезпечення.

8. Відсутній або недостатній попередній аналіз предметної площини та бізнес-процесів. Однією з причин виникнення помилок у програмних системах є відсутність або поверхневий характер попереднього аналізу предметної області та бізнес-процесів. Недостатня увага до етапу формалізації вимог призводить до низки серйозних проблем: некоректного моделювання сутностей, хибного визначення допустимих діапазонів значень, ігнорування критичних станів системи. У результаті логіка реалізації ґрунтується не на перевірених специфікаціях, а на неявних припущеннях розробника.

Такі припущення, сформовані на основі обмеженого розуміння проблемної сфери, часто не витримують перевірки під час роботи з реальними, непередбачуваними або змінюваними даними. Це, у свою чергу, призводить до помилок виконання, порушення цілісності даних та необхідності постійного виправлення коду після введення в експлуатацію.

Для запобігання цим ризикам необхідно впроваджувати чітке формулювання вимог, створення узгоджених специфікацій та системне моделювання бізнес-процесів із залученням експертів предметної області.

Як ми бачимо, деякі проблеми можуть бути вирішеннями налаштуванням компіляторів, деякі – використанням сучасної мови програмування, але деякі –вимагають певної зміни мислення серед розробників [3, 4, 6, 9, 14]. Наприклад, примус до обробки всіх варіантів у match може сприйматись як надлишковий. Водночас це ціна, яку варто платити за надійність і передбачуваність поведінки системи.

Значна частина зазначених проблем пов'язана з відсутністю або недостатністю механізмів статичного аналізу та перевірки повноти логіки на етапі компіляції. Багато мов і компіляторів не здатні автоматично виявити необроблені варіанти, неперевірені параметри, часткові функції чи відсутність обробки крайових умов. Це створює об'єктивну потребу в використанні та інтеграції більш «суворих» інструментів – як-от статичні аналізатори коду, системи контрактного програмування, типи з явною обробкою помилок (Option, Result), тощо. Усвідомлення цього факту є першим кроком до впровадження ефективніших інструментів, практик і методів, які забезпечують надійність цифрових систем ще на етапі розробки.

Висновки. Проблема забезпечення коректності та безпечності програмного забезпечення залишається однією з найактуальніших у сучасній інженерії програмних систем. Як свідчить проведений аналіз, значна частина помилок виникає не стільки через складність алгоритмів або системних вимог, скільки через

нехтування базовими принципами надійної розробки: неповну обробку варіантів, відсутність валідації вхідних параметрів, використання часткових функцій, надмірне покладання на узагальнені типи та обмеження обробки лише основного сценарію виконання..

У статті було проаналізовано низку підходів до підвищення якості програмного забезпечення, які демонструють можливість значного зменшення кількості помилок за рахунок структурного проєктування, формального моделювання та використання сучасних засобів статичного аналізу. Встановлено, що більшість типових помилок можуть бути виявлені ще на ранніх етапах розробки – зокрема, на етапі компіляції – за умови використання мов програмування, які підтримують суворі, але виразні системи типів, а також відповідних налаштувань компіляторів.

Одним із важливих висновків є те, що сучасні компілятори дедалі частіше виконують не лише синтаксичний аналіз, а й семантичну верифікацію, здатні виявляти частковість функцій, відсутність обробки окремих варіантів або некоректні припущення щодо допустимих значень. Це відкриває нові можливості для автоматизованого забезпечення коректності без значного збільшення витрат на ручне тестування або формальну верифікацію.

Водночас для ефективного впровадження зазначених підходів недостатньо лише технологічної підтримки. Необхідною умовою є трансформація самої інженерної культури розробки – з акцентом на проєктування, моделювання предметної області та формалізацію знань. Важливу роль тут відіграє відмова від надмірного узагальнення у типах, активне використання алгебраїчних типів даних, а також практики, що забезпечують повноту та передбачуваність поведінки систем.

Крім того, особливої уваги заслуговує проблема втрати предметно-орієнтованих знань під час переходу від вимог до реалізації. Формальне моделювання, типобезпечна архітектура та декларативний підхід до опису бізнес-логіки здатні істотно зменшити ймовірність виникнення помилок, пов’язаних із невірною інтерпретацією вимог.

Слід також зазначити, що проміжні результати проведеного аналізу були апробовані та представлені автором на профільних науково-технічних конференціях [16–18], що засвідчує практичну релевантність та наукову новизну запропонованих підходів.

Таким чином, підвищення коректності та безпечності програмного забезпечення досягається не лише за рахунок технологічних засобів, а й шляхом поєднання інструментальної підтримки, належного проєктування та зміни підходів до розробки. Сучасні мови програмування – з виразними типами, контролем варіантів і контрактами – у поєднанні з компіляторами нового покоління, створюють основу для значного зростання надійності систем навіть у складних прикладних середовищах.

Перспективним напрямом подальших досліджень є: розробка та систематизація практик використання «розширених» типів і механізмів перевірки варіантів виконання на рівні компіляції; автоматизоване виведення обмежень та інваріантів на основі аналізу вхідних специфікацій; формалізація предметно-орієнтованих знань за допомогою типових систем із підтримкою семантичних обмежень; застосування категорійних моделей для опису загальних структур коректних програм; побудова адаптивних компіляторів, здатних до інтеграції з зовнішніми перевіряльниками або дедуктивними системами.

Отже, забезпечення коректності не слід розглядати як виключно технічну проблему – це інтегральна складова сучасних практик розробки програмного забезпечення, що потребує міждисциплінарного осмислення, методологічного апарату і вдосконалення інструментів програмної інженерії.

СПИСОК ВИКОРИСТАНОЇ ЛІТЕРАТУРИ:

1. Adkins H., Beyer B., Blankinship P., Lewandowski P., Oprea A., Stubblefield A. Building secure and reliable systems: best practices for designing, implementing, and maintaining systems. 1st ed. Sebastopol : O’Reilly Media, 2020. 555 p.
2. Booch G. Object-Oriented Analysis and Design with Applications. 3rd ed. Boston : Addison-Wesley Professional, 2007. 720 p
3. Brooks F. The mythical man-month: essays on software engineering. Anniversary ed. Boston : Addison-Wesley Professional, 1995. 336 p.
4. Evans E. Domain-driven design: tackling complexity in the heart of software. 1st ed. Boston : Addison-Wesley Professional, 2003. 560 p.
5. Forsgren N., Humble J., Kim G. Accelerate: the science of lean software and DevOps: building and scaling high performing technology organizations. Illustrated ed. Portland : IT Revolution, 2018. 288 p.
6. Galef J. The Scout Mindset: Why Some People See Things Clearly and Others Don’t. New York : Portfolio / Penguin, 2021. 288 p.
7. Gamma E., Helm R., Johnson R., Vlissides J. Design patterns: elements of reusable object-oriented software. 1st ed. Boston : Addison-Wesley Professional, 1994. 416 p.
8. Herbert K., Plante D. Refining Existing Types in Scala With Refined. https://www.baeldung.com/scala/refined-types
9. Hunt A., Thomas D. The pragmatic programmer: from journeyman to master. Boston : Addison-Wesley Professional, 1999. 352 p.
10. Martin R. Clean code: a handbook of agile software craftsmanship. 1st ed. Boston : Pearson, 2008. 464 p.
11. Martin R. Functional design: principles, patterns, and practices. 1st ed. Boston : Addison-Wesley Professional, 2023. 384 p.
12. Ousterhout J. A philosophy of software design. 1st ed. Stanford : Yaknyam Press, 2018. 190 p.
13. Pilquist M., Chiusano P., Bjarnasson R. Functional programming in Scala. 2nd ed. Shelter Island : Manning, 2023. 488 p.
14. Winitzki S. The Science of Functional Programming. Part I. Raleigh : Lulu Press, 2021. 280 p.
15. Winters T., Manshreck T., Wright H. Software engineering at Google: lessons learned from programming over time. 1st ed. Sebastopol : O’Reilly Media, 2020. 599 p.
16. Рихальський О. Ю. Методи підвищення коректності програмного забезпечення: сучасний функціонал компіляторів та інші підходи // VII Всеукраїнська науково-технічна конференція «Комп’ютерні технології: інновації, проблеми, рішення», 02–03 грудня 2024 р. Житомир, Україна.
17. Рихальський О. Ю. Підвищення коректності програмного забезпечення методом звуження областей визначення // Всеукраїнська науково-технічна інтернет-конференція «Автоматизація, комп’ютерні та біомедичні технології», 26 березня 2025 р. Дніпро, Україна.
18. Рихальський О. Ю. Шляхи підвищення кіберстійкості програмного забезпечення: сучасні мови програмування та супутні методології // XI Міжнародна науково-практична конференція «Актуальні питання забезпечення кібербезпеки та захисту інформації», 24 квітня 2025 р. Київ, Україна.

REFERENCES:

1. Adkins, H., Beyer, B., Blankinship, P., Lewandowski, P., Oprea, A., & Stubblefield, A. (2020). *Building secure and reliable systems: Best practices for designing, implementing, and maintaining systems*. O'Reilly Media.

2. Booch, G. (2007). *Object-oriented analysis and design with applications (3rd ed.)*. Addison-Wesley Professional.

3. Brooks, F. P. (1995). *The mythical man-month: Essays on software engineering (Anniversary ed.)*. Addison-Wesley Professional.

4. Evans, E. (2003). *Domain-driven design: Tackling complexity in the heart of software*. Addison-Wesley Professional.

5. Forsgren, N., Humble, J., & Kim, G. (2018). *Accelerate: The science of lean software and DevOps: Building and scaling high performing technology organizations (Illustrated ed.)*. IT Revolution.

6. Galef, J. (2021). *The scout mindset: Why some people see things clearly and others don't.* Portfolio / Penguin.

7. Gamma, E., Helm, R., Johnson, R., & Vlissides, J. (1994). *Design patterns: Elements of reusable object-oriented software*. Addison-Wesley Professional.

8. Herbert, K., & Plante, D. (n.d.). *Refining existing types in Scala with Refined*. Baeldung. https://www.baeldung.com/scala/refined-types

9. Hunt, A., & Thomas, D. (1999). *The pragmatic programmer: From journeyman to master*. Addison-Wesley Professional.

10. Martin, R. C. (2008). *Clean code: A handbook of agile software craftsmanship*. Pearson.

11. Martin, R. C. (2023). *Functional design: Principles, patterns, and practices*. Addison-Wesley Professional.

12. Ousterhout, J. (2018). *A philosophy of software design (1st ed.)*. Yaknyam Press.

13. Pilquist, M., Chiusano, P., & Bjarnason, R. (2023). *Functional programming in Scala (2nd ed.)*. Manning.

14. Winitzki, S. (2021). *The science of functional programming. Part I*. Lulu Press.

15. Winters, T., Manshreck, T., & Wright, H. (2020). *Software engineering at Google: Lessons learned from programming over time (1st ed.)*. O'Reilly Media.

16. Rykhalskyy, O. Y. (2024, December 2–3). Metody pidvyshchennia korektnosti prohramnoho zabezpechennia: suchasnyi funktsional kompiliatoriv ta inshi pidkhody [Methods for improving software correctness: Modern compiler functionality and other approaches]. Proceedings of the *VII Vseukrainska naukovo-tekhnichna konferentsiia «Kompiuterni tekhnolohii: innovatsii, problemy, rishennia». Zhytomyr, Ukraine*.

17. Rykhalskyy, O. Y. (2025, March 26). Pidvyshchennia korektnosti prohramnoho zabezpechennia metodom zvuzhennia oblastei vyznachennia [Improving software correctness by narrowing the domains]. Proceedings of the *Vseukrainska naukovo-tekhnichna internet-konferentsiia "Avtomatyzatsiia, kompiuterni ta biomedychni tekhnolohii". Dnipro, Ukraine*.

18. Rykhalskyy, O. Y. (2025, April 24). Shliakhy pidvyshchennia kiberstiikosti prohramnoho zabezpechennia: suchasni movy prohramuvannia ta suputni metodolohii [Ways to enhance software cyber resilience: Modern programming languages and related methodologies]. Proceedings of the *XI Mizhnarodna naukovo-praktychna konferentsiia "Aktualni pytannia zabezpechennia kiberbezpeky ta zakhystu informatsii". Kyiv, Ukraine*.