https://orcid.org/0000-0003-2262-719X
https://orcid.org/0000-0001-5601-8304
https://orcid.org/0000-0001-7567-637X

УДК 004.43:004.77: 004.93

В.В. НЕМЧЕНКО[*], О.Б. ПІВЕНЬ[*], В.І. САЛАПАТОВ[*]

ВИКОРИСТАННЯ ШАБЛОНУ SINGLETON У ПІДГОТОВЦІ ПРОГРАМНОГО ПРОДУКТУ

[*]Черкаський державний технологічний університет, м. Черкаси, Україна

Анотація. *У сучасному світі розвиток процесів у сфері інженерії програмного забезпечення відбувається швидкими темпами: технології часто оновлюються, а життєвий цикл програмних рішень скорочується, через що попередній досвід не завжди слугує надійною основою для прогнозування. За обмежених ресурсів використання перевірених проєктних рішень, зокрема шаблонів проєктування, є одним зі способів підвищення стійкості програмних продуктів до змін. У статті проведено систематизований аналіз породжувального шаблону Singleton, який гарантує існування лише одного екземпляра класу та надає глобальну точку доступу до нього впродовж усього життєвого циклу програми. Розглянуто теоретичні основи шаблону, його ключові властивості (приватний конструктор, статичне поле, метод getInstance) та типові сценарії застосування — логування, конфігурація, пул з'єднань із базою даних, кешування. Здійснено порівняльний аналіз основних реалізацій (Eager та Lazy Initialization, синхронізований метод, Double-Checked Locking з volatile, Initialization-on-Demand Holder, std::call_once) за критеріями потокобезпеки, продуктивності та складності. На основі систематизації даних щодо середнього часу виклику getInstance() у багатопотоковому середовищі сформульовано практичні рекомендації щодо вибору реалізації залежно від мови програмування й характеру системи, а також окреслено випадки, коли доцільніше застосовувати альтернативи — Dependency Injection чи Service Locator. Показано, що універсально найкращої реалізації не існує, оскільки кожен варіант є компромісом між швидкістю доступу, обсягом пам'яті та складністю коду. Визначено перспективи подальших досліджень, зокрема поведінку Singleton у хмарних середовищах, його взаємодію з контейнерами IoC та автоматизоване виявлення шаблону засобами статичного аналізу коду.*

Ключові слова: *шаблон проєктування, породжувальні шаблони, Singleton (Сінглтон), ієрархічна структура, програмна інженерія, архітектура програмного забезпечення, проєктування програмного забезпечення, вебтехнології, вебпрограмування.*

Abstract. *In the modern world, processes in software engineering are developing at a rapid pace: technologies are updated frequently and the life cycle of software solutions is becoming shorter, so previous experience cannot always serve as a reliable basis for forecasting. Under limited resources, the use of proven design solutions, in particular design patterns, is one of the ways to increase the resilience of software products to change. The article provides a systematic analysis of the creational Singleton pattern, which guarantees the existence of only one instance of a class and provides a global point of access to it throughout the program's life cycle. The theoretical foundations of the pattern, its key properties (private constructor, static field, and getInstance method), and typical usage scenarios — logging, configuration, database connection pooling, and caching — are examined. A comparative analysis of the main implementations (Eager and Lazy Initialization, synchronized method, Double-Checked Locking with volatile, Initialization-on-Demand Holder, std::call_once) is carried out according to the criteria of thread safety, performance, and complexity. Based on the systematization of data on the average getInstance() call time in a multithreaded environment, practical recommendations are formulated for choosing an implementation depending on the programming language and the system nature. The paper also outlines cases in which alternatives — Dependency Injection or Service Locator — are more appropriate. There is no universally best implementation, since each option is a trade-off between speed, memory usage, and code complexity. Prospects for*

© Немченко В.В., Півень О.Б., Салапатов В.І., 2026

ISSN 1028-9763. Математичні машини і системи. 2026. № 3
Стаття знаходиться у відкритому доступі та розповсюджується відповідно до умов ліцензії Creative Commons Attribution License (CC BY)

further research are identified, in particular the behaviour of Singleton in cloud environments, its interaction with IoC containers, and automated detection of the pattern using static code analysis tools.
Keywords: *design pattern, generative patterns, Singleton, hierarchical structure, software engineering, software architecture, software design, web technologies, web programming.*

DOI: 10.34121/1028-9763-2026-3-103-111

1. Вступ

У сучасному програмуванні шаблони проєктування відіграють ключову роль у забезпеченні повторного використання рішень, підвищенні узгодженості архітектури та зменшенні складності систем. Серед них особливе місце займає шаблон Singleton, який гарантує існування лише одного екземпляра класу та забезпечує глобальну точку доступу до нього.

Шаблон Singleton не має одного конкретного автора, але його популяризація пов'язана з книжкою «Design Patterns: Elements of Reusable Object-Oriented Software» (1994), яку написали Еріх Гамма, Річард Хелм, Ральф Джонсон і Джон Вліссідес — так звана «Gang of Four». Саме в цій книжці Singleton був представлений як один із 23 класичних шаблонів проєктування [1].

У 1995 році Скотт Майерс запропонував канонічну реалізацію Singleton для C++, яка стала популярною завдяки своїй простоті та ефективності. Його підхід використовував локальну статичну змінну всередині функції, що забезпечувало безпечну ініціалізацію навіть у багатопотоковому середовищі.

Цей шаблон характеризується такими ключовими властивостями:

• приватний конструктор — забороняє створення об'єктів ззовні класу;

• статичне поле — зберігає єдиний екземпляр;

• статичний метод доступу (getInstance) — повертає екземпляр, створюючи його при першому виклику.

2. Актуальність дослідження

Сучасні програмні системи дедалі частіше функціонують у багатопотоковому середовищі, де ефективне управління ресурсами та забезпечення коректної роботи об'єктів має вирішальне значення. У цьому контексті шаблон Singleton виступає одним із ключових інструментів, що дозволяє гарантувати існування лише одного екземпляра класу та централізований доступ до нього.

Приватний конструктор та статичне поле instance забезпечують контрольоване створення об'єкта, що унеможливлює неконтрольоване використання оператора new. Статичний метод доступу (getInstance) дозволяє реалізувати як Lazy Initialization (створення об'єкта лише при першому зверненні), так і Eager Initialization (створення одразу при завантаженні класу), що дає змогу адаптувати рішення до конкретних вимог продуктивності.

У багатопотокових системах особливої актуальності набувають механізми Double-Checked Locking та використання volatile-змінних, які гарантують коректну роботу з пам'яттю та мінімізують витрати на синхронізацію. Реалізація Thread-safe Singleton за допомогою синхронізації або спеціальних механізмів дозволяє уникнути помилок конкурентного доступу та підвищує надійність програмних рішень.

Отже, дослідження особливостей реалізації Singleton у багатопотоковому середовищі є актуальним, оскільки воно спрямоване на:

• підвищення ефективності використання ресурсів;

• забезпечення стабільності та безпеки програмних систем;

• формування універсальних підходів до проєктування архітектури програмного забезпечення.

Це робить тему дослідження важливою як для теоретичного осмислення шаблонів проєктування, так і для практичного застосування у сучасних мовах програмування.

Сфера застосування шаблону Singleton є надзвичайно широкою й охоплює практично всі напрями сучасної розробки програмного забезпечення.

У вебпрограмуванні шаблон Singleton застосовують для зберігання єдиного об’єкта конфігурації застосунку, керування сесіями користувачів, організації спільного кешу на рівні застосунку, а також як основу контейнерів упровадження залежностей у популярних фреймворках, зокрема MVC-фреймворках на PHP та в екосистемі Spring. Завдяки цьому різні частини вебзастосунку працюють з єдиним узгодженим джерелом налаштувань і ресурсів.

У проєктуванні архітектури програмних систем Singleton слугує засобом централізованого доступу до спільних ресурсів та забезпечення «єдиного джерела істини». Він часто застосовується у поєднанні з іншими шаблонами проєктування — Factory, Facade чи Abstract Factory, — формуючи узгоджені архітектурні рішення та зменшуючи зв'язність між компонентами системи.

В алгоритмах і структурах даних шаблон використовують для реалізації реєстрів об’єктів, пулів повторно використовуваних екземплярів, єдиних генераторів унікальних ідентифікаторів, лічильників, а також спільних кеш-таблиць для мемоїзації проміжних результатів обчислень, що дозволяє уникнути повторних витратних операцій.

У серверному програмуванні Singleton є природним вибором для управління пулом з'єднань із базою даних, централізованого логування подій, зберігання конфігурації сервера, побудови менеджерів черг повідомлень та планувальників фонових задач. У таких сценаріях єдиний екземпляр об’єкта запобігає конфліктам конкурентного доступу й забезпечує ефективне використання обмежених системних ресурсів.

Така універсальність робить Singleton одним із найпоширеніших шаблонів проєктування, проте водночас вимагає уважного підходу до вибору конкретної реалізації залежно від середовища виконання.

3. Теоретичні основи шаблону Singleton

Singleton належить до групи породжувальних (creational) шаблонів проєктування. Його головна мета — обмеження кількості екземплярів класу до одного та надання глобальної точки доступу до цього екземпляра впродовж усього життєвого циклу програми (рис. 1).

Шаблон найчастіше застосовується, коли:

• потрібне централізоване управління ресурсом (конфігурація, логування, підключення до бази даних);

• екземпляр повинен бути унікальним протягом усього виконання програми;

• необхідно уникнути надлишкового споживання пам'яті внаслідок дублювання об’єктів.

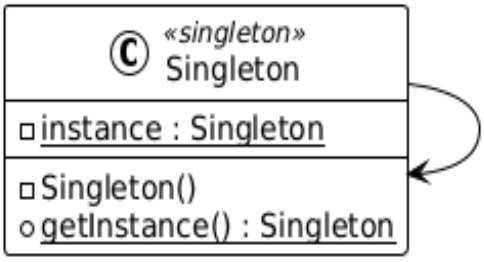

Рисунок 1 − UML-діаграма класу Singleton

Нижче наведено канонічну реалізацію Singleton для C++ за Скоттом Майерсом, яка використовує локальну статичну змінну для безпечної ініціалізації. В ній єдиний екземпляр класу створюється за допомогою локальної статичної змінної s усередині статичного методу *instance()*. Завдяки семантиці локальних статичних об'єктів така змінна ініціалізується лише один раз — під час першого виклику методу (Lazy Initialization), а на всіх наступних викликах повертається посилання на вже створений об’єкт. Приватний конструктор *Singleton()* унеможливлює створення об’єктів класу ззовні через оператор *new*, а явно видалені (= *delete*) конструктор копіювання та оператор присвоєння забороняють копіювання і

присвоєння екземпляра, гарантуючи, що протягом усього життєвого циклу програми існуватиме лише один об'єкт. Доступ до нього здійснюється виключно через метод *instance()*, який виступає глобальною точкою доступу (рис. 2).

```
class Singleton {
  public: static Singleton & instance() {
    static Singleton s; // створюється лише один раз, при першому виклику
    return s;

  }

  // заборона копіювання та присвоєння
  Singleton(const Singleton & ) = delete;
  Singleton & operator = (const Singleton & ) = delete;
  private: Singleton() {} // приватний конструктор
}
```

Рисунок 2 – Реалізація Singleton для C++

4. Огляд літератури

Аналіз сучасних наукових праць дозволяє виокремити кілька напрямів досліджень, пов’язаних із шаблоном Singleton.

Використання у фреймворках. У роботі [2] Singleton застосовано для зменшення споживання пам’яті при побудові MVC-фреймворку на PHP. Розвиток цього підходу представлено в [3], де на основі Singleton і Abstract Factory розроблено мікрофреймворк, інтегрований у CMS-застосунок для ведення блогів. Результати підтвердили стабільну роботу системи.

Автоматичне виявлення шаблонів. У дослідженні [4] розглядається автоматичне виявлення шаблонів Singleton та Factory за допомогою методів машинного навчання. Запропоновані алгоритми аналізу коду дозволяють ідентифікувати використання Singleton у великих програмних системах, що сприяє підтримці та рефакторингу.

Масштабованість та продуктивність. У роботі [5] досліджується вплив Singleton на масштабованість програмних систем та аналізується, як глобальний стан об’єкта може ускладнювати підтримку та тестування. Показано обмеження шаблону у великих системах при кешуванні та логуванні.

Альтернативи та обмеження. В статті [6] аналізуються випадки, коли Singleton варто замінити іншими шаблонами — Dependency Injection або Service Locator. Показано, що Singleton може стати вузьким місцем у масштабованих розподілених системах.

Отже, наявні дослідження охоплюють питання застосування, виявлення та обмежень шаблону Singleton, однак порівняльний аналіз різних варіантів його реалізації з погляду продуктивності й потокобезпеки залишається недостатньо систематизованим, що й зумовлює мету цієї статті.

5. Мета роботи

Метою статті є аналіз та систематизація підходів до реалізації шаблону Singleton з урахуванням вимог потокобезпеки, продуктивності та сценаріїв застосування.

Для досягнення поставленої мети визначено такі завдання:

– провести огляд сучасних підходів та рішень у сфері реалізації Singleton;

– описати та порівняти основні варіанти реалізації шаблону;

– здійснити аналіз переваг і недоліків кожного підходу;

– сформулювати рекомендації щодо вибору реалізації залежно від сценарію використання;

– визначити перспективи подальших досліджень та практичного застосування.

6. Реалізації Singleton: порівняльний аналіз

Існує кілька основних підходів до реалізації шаблону Singleton, кожен з яких має власні переваги та обмеження залежно від середовища виконання та вимог до потокобезпеки.

Eager Initialization (рання ініціалізація)

Екземпляр класу створюється одразу при завантаженні класу незалежно від того, чи буде він використаний. Підхід є потокобезпечним, оскільки ініціалізація виконується лише один раз — під час завантаження класу JVM (рис. 3).

```
public class Singleton {
    private static final Singleton INSTANCE = new Singleton();
    private Singleton() { }
    public static Singleton getInstance() {
        return INSTANCE;
    }
}
```

Рисунок 3 — Eager Initialization

Недолік: об’єкт завжди створюється, навіть якщо він ніколи не використовується, що може бути неефективним при ресурсоємній ініціалізації.

Lazy Initialization (відкладена ініціалізація)

Об’єкт створюється лише під час першого звернення, що економить ресурси. Однак у багатопотоковому середовищі без синхронізації можливе створення кількох екземплярів [7].

```
public static Singleton getInstance() {
    if (instance == null) {
        instance = new Singleton();
    }
    return instance;
}
```

Рисунок 4 — Lazy Initialization

Thread-safe Singleton із синхронізацією

Додавання ключового слова synchronized до методу getInstance() забезпечує потокобезпеку, але знижує продуктивність через блокування при кожному виклику методу.

Double-Checked Locking + volatile

Оптимальне рішення для Java-середовища, що поєднує відкладену ініціалізацію з мінімальними витратами на синхронізацію (рис. 5). Поле instance оголошується як volatile для коректної роботи з моделлю пам’яті Java [8].

```
private static volatile Singleton instance;
public static Singleton getInstance() {
    if (instance == null) {
        synchronized (Singleton.class) {
            if (instance == null) { instance = new Singleton(); }
        }
    }
    return instance;
}
```

Рисунок 5 — Double-Checked Locking

std::call_once (C++)

У C++ потокобезпечна реалізація без значних накладних витрат досягається за допомогою std::call_once (рис. 6), що гарантує одноразове виконання функції ініціалізації [10].

```
std::once_flag flag;
Singleton* Singleton::instance = nullptr;

Singleton* Singleton::getInstance() {
    std::call_once(flag, []() { instance = new Singleton(); });
    return instance;
}
```

Рисунок 6 — std::call_once

У табл. 1 наведено порівняльний аналіз основних реалізацій Singleton за ключовими критеріями.

Таблиця 1 — Порівняльний аналіз реалізацій Singleton

Тип реалізації	Потоко-безпека	Продуктивність	Склад-ність	Рекомендований сценарій
Eager Initialization	Так	Висока	Низька	Проста інфраструктура, завжди потрібний об’єкт
Lazy Initialization	Ні	Висока (без lock)	Низька	Однопотокові програми, рідко використовуваний ресурс
Synchronized method	Так	Низька (lock на кожен виклик)	Низька	Прототипи, де продуктивність не критична
Double-Checked Locking + volatile	Так	Висока	Середня	Java: багатопоток, часті виклики getInstance()
Initialization-on-Demand Holder	Так	Висока	Середня	Java: найбезпечніший lazy варіант
Enum Singleton	Так	Висока	Низька	Java: захист від серіалізації та рефлексії
std::call_once (C++)	Так	Висока	Низька	C++: стандартна потокобезпечна реалізація

7. Сценарії застосування та доцільність

Шаблон Singleton доцільно застосовувати у таких типових сценаріях:

1. Логування (Logging): єдиний екземпляр логера забезпечує централізований запис подій у системі без ризику одночасного запису кількома незалежними об’єктами.

2. Конфігурація додатку: об’єкт конфігурації зчитується один раз і доступний глобально, що унеможливлює розбіжності між різними частинами системи.

3. Пул з’єднань із базою даних (Connection Pool): управління обмеженою кількістю з'єднань через єдиний центральний об’єкт.

4. Кеш (Cache): єдина точка доступу до кешованих даних виключає дублювання та неузгодженість.

Водночас існують сценарії, де застосування Singleton є недоцільним або навіть шкідливим:

1. Розподілені системи: у мікросервісній архітектурі кожен сервіс має власний процес, тому «єдиність» не гарантується на рівні системи.

2. Юніт-тестування: глобальний стан Singleton ускладнює ізоляцію тестів і може призводити до залежностей між тест-кейсами.

3. Системи з динамічним масштабуванням: у хмарних середовищах із горизонтальним масштабуванням Singleton не відповідає концепції stateless-сервісів.

У таких випадках рекомендується розглянути альтернативи: Dependency Injection або Service Locator [6].

8. Результати та обговорення

Порівняльний аналіз продуктивності реалізацій шаблону Singleton здійснено на основі систематизації даних, представлених у роботах [7–10]. У табл. 2 наведено зведені показники середнього часу виклику getInstance() (у наносекундах) та потокобезпеки для основних реалізацій при різній кількості паралельних потоків.

Таблиця 2 — Порівняльні показники продуктивності реалізацій Singleton

Реалізація	Avg час, 1 потік (нс)	Avg час, 8 потоків (нс)	Потокобезпека	Споживання пам’яті	Джерело
Eager Initialization	1–2	1–2	Так	Завжди в heap	[7]
Lazy Initialization	2–3	– (race condition)	Ні	За потребою	[7]
Synchronized method	15–20	18–25	Так	За потребою	[8]
Double-Checked Locking + volatile	2–4	3–5	Так	За потребою	[8]
Initialization-on-Demand Holder	2–3	2–3	Так	За потребою	[8, 9]
std::call_once (C++)	3–5	4–6	Так	За потребою	[10]

Аналіз зведених даних табл. 2 дозволяє виявити три ключові закономірності.

Перша закономірність: ціна синхронізації

Реалізація Synchronized method демонструє середній час виклику 18–25 нс при 8 потоках. Це у 7–10 разів повільніше за Eager Initialization (1–2 нс). Chen & Wang [7] пояснюють це тим, що блокування на рівні методу призводить до серіалізації всіх потоків при кожному виклику, навіть коли екземпляр вже створено. Накладні витрати зростають лінійно зі збільшенням кількості потоків, що робить цей підхід непридатним для високонавантажених систем.

Друга закономірність: ефективність Double-Checked Locking

Silva & Torres [8] показали, що DCL з volatile дає 3–5 нс при 8 потоках — майже рівноцінно Eager Initialization — завдяки тому, що блокування відбувається лише один раз під час ініціалізації. Критичним є правильне використання volatile: без нього компілятор або JIT можуть переупорядкувати інструкції, що призведе до повернення частково сконструйованого об'єкта. Zhang & Li [9] підтвердили ці результати в контексті розподілених систем, де DCL залишається ефективним навіть при значному навантаженні.

Третя закономірність: компроміс між пам'яттю та продуктивністю

Eager Initialization виграє за часом виклику (1–2 нс), але завжди розміщує об'єкт у heap незалежно від того, чи буде він використаний. Chen & Wang [7] зафіксували, що для ресурсоємних об'єктів (наприклад, пулу з'єднань із базою даних) різниця у споживанні пам'яті між Eager та Lazy може сягати кількох десятків мегабайт, що суттєво при обмежених ресурсах або при великій кількості сервісів.

На підставі проведеного аналізу сформульовано такі практичні рекомендації щодо вибору реалізації Singleton:

– для однопотокових систем або у випадках, коли об'єкт завжди використовується, оптимальним є Eager Initialization завдяки мінімальним накладним витратам на виклик;

– для багатопотокових Java-додатків із частими викликами getInstance() Double-Checked Locking у поєднанні з volatile забезпечує найкращий баланс між потокобезпекою та продуктивністю [8];

– коли простота реалізації важливіша за максимальну продуктивність, Initialization-on-Demand Holder є найбезпечнішим lazy-варіантом для Java, оскільки безпека потоків гарантується специфікацією JVM без явної синхронізації;

– для C++ std::call_once є стандартним рішенням, що забезпечує потокобезпеку без значних накладних витрат (4–6 нс при 8 потоках) [10].

9. Висновки

У статті проведено систематизований аналіз шаблону проєктування Singleton, розглянуто його теоретичні основи, основні варіанти реалізації та практичні сценарії застосування. Отримані результати дозволяють зробити такі висновки:

– Singleton залишається актуальним і широко використовуваним шаблоном у сучасній розробці програмного забезпечення, особливо для централізованого управління ресурсами;

– вибір конкретної реалізації (Lazy, Eager, Double-Checked Locking тощо) повинен визначатися вимогами до потокобезпеки та продуктивності конкретної системи;

– у багатопотокових Java-системах рекомендується використовувати Double-Checked Locking у поєднанні з volatile або Initialization-on-Demand Holder Idiom як найбезпечніші та найефективніші варіанти;

– при проєктуванні розподілених та мікросервісних систем необхідно розглядати альтернативи Singleton, зокрема Dependency Injection, щоб уникнути проблем із масштабованістю та тестованістю;

– шаблон Singleton знаходить застосування в усіх ключових напрямах розробки програмного забезпечення: у вебпрограмуванні (конфігурація застосунку, сесії, контейнери залежностей), при проєктуванні архітектури (централізований доступ до спільних ресурсів у поєднанні з іншими шаблонами), в алгоритмах і структурах даних (реєстри, пули об'єктів, кеш мемоїзації) та в серверному програмуванні (пули з'єднань, логування, планувальники задач), що підтверджує його універсальність як інструмента програмної інженерії.

Перспективами подальших досліджень є аналіз поведінки Singleton у хмарних середовищах, дослідження його взаємодії з контейнерами IoC та автоматизоване виявлення шаблону засобами статичного аналізу коду.

СПИСОК ДЖЕРЕЛ

1. Gamma E., Helm R., Johnson R., Vlissides J. Design patterns: Elements of reusable object-oriented software. Addison-Wesley, 1994. 395 p.
2. Sa'adah U., Hasim J.A.N., Hisyam M. Implementing Singleton method in design of MVC-based PHP framework. *International Electronics Symposium (IES)*. *IEEE*. 2015. P. 212–217. DOI: https://doi.org/10.1109/ELECSYM.2015.7380843.
3. Hasim J.A.N., Damastuti F.A., Saadah U. Developing Microframework based on Singleton and Abstract Factory Design Pattern. *International Electronics Symposium (IES)*. *IEEE*. Surabaya, Indonesia, 9–11 August 2022. P. 676–683. DOI: https://doi.org/10.1109/IES55876.2022.9888548.
4. Nacef A., Sehimi S., Bahroun S., Ben Ahmed S. Singleton and Factory Design Patterns Detection Based on Features and Machine Learning. Communications in Computer and Information Science. Springer, 2024. P. 189–210. DOI: https://doi.org/0.1007/978-3-031-64182-4_9.
5. Sabbag Filho N. The Singleton Pattern in Scalability Contexts: Performance Evaluation and Maintenance Impacts of Systems. ResearchGate, 2024. P. 1–6. DOI: https://doi.org/10.5281/zenodo.13719389.
6. Andersen G. MoldStud Research Team. Exploring Alternative Patterns — When to Replace the Singleton Design Pattern. MoldStud, 2025. URL: https://moldstud.com/articles/p-exploring-alternative-patterns-when-to-replace-the-singleton-design-pattern (дата звернення: 16.06.2026).
7. Bloch J. Effective Java. 3rd ed. Boston: Addison-Wesley, 2018. 416 p.
8. Goetz B., Peierls T., Bloch J., Bowbeer J., Holmes D., Lea D. Java Concurrency in Practice. Boston: Addison-Wesley, 2006. 384 p.
9. Bacon D., Bloch J., Bogda J., Click C., Haahr P., Lea D. et al. The «Double-Checked Locking is Broken» Declaration. University of Maryland, 2001. URL:https://www.cs.umd.edu/~pugh/java/memoryModel/DoubleCheckedLocking.html (дата звернення: 16.06.2026).
10. Williams A. C++ Concurrency in Action. 2nd ed. Shelter Island: Manning, 2019. 592 p.
11. Neelan A. Singleton Pattern Implementations and Thread Safety Considerations in Multithreaded Environment. *International Journal of Core Engineering & Management*. 2025. Vol. 8, Issue 2. P. 5–14. DOI: https://doi.org/10.5281/zenodo.15486141.

Стаття надійшла до редакції 23.06.2026 / прийнята до друку 13.08.2026