«Системні технології» 1 (162) 2026 «System technologies»
DOI 10.34185/1562-9945-5-162-2026-15
УДК 004.021; 004.383

Н.О. Матвєєва

ПОРІВНЯЛЬНИЙ АНАЛІЗ ЕФЕКТИВНОСТІ ЗАСТОСУВАННЯ ПАТЕРНІВ

Анотація. Розглянуто та систематизовано особливості реалізації основних породжувальних патернів мовами програмування C#, C++, Python та Java. Проведено порівняльний аналіз реалізацій патернів. Визначено спільні риси та відмінності у підходах до реалізації патернів залежно від рівня типізації та механізмів керування пам'яттю.
Ключові слова: патерн, Factory Method, Abstract Factory, Builder, Prototype, Singleton, порівняльний аналіз, реалізація, мови програмування.

Вступ. Одним із ключових інструментів об'єктно-орієнтованого проєктування є патерни проєктування, які являють собою формалізовані, перевірені практикою рішення типових архітектурних проблем. Особливе місце серед них займають породжувальні патерни проєктування, основним завданням яких є абстрагування процесу створення об'єктів та зменшення залежностей між компонентами системи. Використання породжувальних патернів дозволяє уникати жорсткої прив'язки коду до конкретних класів, спрощує розширення системи та підвищує її стійкість до змін.

Актуальність дослідження патернів проєктування обумовлена тим, що саме процес створення об'єктів часто стає джерелом архітектурних проблем у складних програмних системах. Некоректно організований механізм ініціалізації об'єктів може призводити до надмірної зв'язаності компонентів, дублювання коду, ускладнення тестування та зниження якості програмного забезпечення загалом.

Використання патернів часто викликає труднощі, пов'язані з вибором конкретного патерна, особливостями реалізації різними мовами програмування та оцінкою доцільності застосування в реальних проєктах. Особливо це актуально в умовах багатомовних проєктів, коли одна й та сама архітектурна ідея реалізується засобами мов із різними парадигмами, моделями типізації та механізмами керування пам'яттю.

Метою статті є розгляд особливостей реалізації основних породжувальних патернів, виконання порівняльного аналізу, а також оцінювання впливу мовних засобів на архітектурні рішення та якість програмного коду.

Основна частина. Вперше патерни описали Gang of Four (GoF)[1]. Саме після їхньої роботи патерни почали входити до університетських курсів, стандартів інженерії програмного забезпечення та корпоративних практик.

Необхідним етапом проведення порівняльного аналізу є обґрунтований вибір середовищ програмування та інструментів розробки, які забезпечують стабільність, відт-

ISSN 1562-9945 (Print)
ISSN 2707-7977 (Online)

ворюваність результатів і зручність аналізу. В роботі використовувались два інтегрованих середовища розробки (IDE), які є де-факто стандартами для відповідних мов програмування [2-5].

Для реалізації та тестування патернів мовами C#, C++ та Python обрано середовище Microsoft Visual Studio, а для мови Java - середовище IntelliJ IDE.

Для забезпечення коректності та відтворюваності результатів використовувалися стандартні бібліотеки мов програмування, без залучення сторонніх фреймворків, щоб уникнути впливу зовнішніх оптимізацій; уніфіковано логіку прикладів, аби всі реалізації патернів виконували однакові дії та мали порівнювану складність; використано вбудовані засоби вимірювання часу виконання, доступні у відповідних мовах програмування; забезпечено однакові умови запуску програм, зокрема запуск у режимі Release (де це можливо) та вимкнення зайвих фонових процесів; проведено попереднє тестування середовищ, щоб переконатися у коректності налаштувань і стабільності виконання [5].

Детально проаналізовано п’ять ключових породжувальних патернів: Factory Method, Abstract Factory, Builder, Prototype та Singleton. Для кожного з них визначено сферу застосування, переваги, обмеження та типові сценарії використання у сучасних програмних системах. Для забезпечення коректності та об’єктивності аналізу всі патерни реалізовувалися з однаковою логічною структурою, що дозволило безпосередньо порівнювати отримані рішення. Реалізації були виконані з урахуванням особливостей кожної мови програмування, зокрема моделей типізації, синтаксичних конструкцій, механізмів керування пам’яттю та стандартних бібліотек.

Для порівняльного аналізу оцінювання здійснювалось за такими критеріями, як:

- кількість рядків коду (LOC);
- складність написання та обсяг шаблонного коду;
- типізація та її вплив на архітектурну строгість;
- зручність реалізації та читабельність коду;
- потенційна ефективність виконання та особливості компіляції або запуску.

Першим розглядався патерн *Factory Method,* для його об’єктивного аналізу використовувалась однакова логічна реалізація, яка створювала два продукти (ProductA, ProductB), застосовувала абстрактний Creator, об’єкти створювалися через перевизначений factory method, виконувала однакову кількість операцій. Результати порівняння наведені у таблицях 1 та 2.

Таблиця 1

Загальне порівняння

Мова	LOC	Boilerplate code	Складність написання	Коментар
C++	~95	Високий	Висока	Найбільший контроль, але складний
Java	~90	Високий	Середня	Класичний GoF-еталон
C#	~70	Середній	Середньо-низька	Синтаксичні спрощення
Python	~50	Низький	Низька	Найкоротший і найгнучкіший

Таблиця 2

Мова	Вплив на патерн
C++	Патерн потрібний для контролю
Java	Патерн — архітектурний стандарт
C#	Патерн можна спростити
Python	Патерн часто концептуальний

Для забезпечення коректності та відтворюваності результатів аналізу використано ідентичну логічну структуру патерна *Abstract Factory* для всіх чотирьох мов програмування[6-7]. Реалізація базується на класичному GoF-підході та має такі спільні характеристики: визначено два типи абстрактних продуктів (ProductA, ProductB); реалізовано два сімейства конкретних продуктів (Family 1 та Family 2); створено абстрактну фабрику з методами створення кожного типу продуктів; клієнтський код працює виключно з абстракціями, без знання конкретних класів; заміна фабрики повністю змінює сімейство створюваних об’єктів без модифікації клієнта. Результати порівняння наведені у таблицях 3 та 4.

Таблиця 3

Загальне порівняння

Мова	LOC (≈)	Boilerplate code	Складність написання	Коментар
C++	~150	Дуже високий	Висока	Максимальна строгість, складне керування пам’яттю
Java	~145	Дуже високий	Середньо-висока	Класичний еталон GoF, добре масштабується
C#	~130	Високий	Середня	Менше шаблонного коду, ніж у Java
Python	~95	Низький	Низька	Найкомпактніша реалізація

Таблиця 4

Мова	Вплив на патенр
C++	Патерн критично важливий для безпеки та контролю
Java	Патерн є архітектурним стандартом
C#	Дозволяє спростити реалізацію без втрати структури
Python	Патерн носить концептуальний характер

Для аналізу патерна *Builder* використано однакову логічну модель побудови складного об’єкта, яка повністю відповідає класичному опису GoF: наявний складний продукт з трьома частинами (PartA, PartB, PartC); визначено інтерфейс Builder, що описує поетапне створення продукту; реалізовано ConcreteBuilder, який інкапсулює процес складання; використано Director, що задає фіксований порядок побудови; клієнт отри-

ISSN 1562-9945 (Print)
ISSN 2707-7977 (Online)

мує готовий продукт лише після завершення конструювання. Результати порівняння наведені у таблицях 5 та 6.

Таблиця 5

Загальне порівняння

Мова	LOC (≈)	Boilerplate code	Складність написання	Коментар
C++	~120	Високий	Висока	Максимальний контроль, мінімальні runtime-витрати
Java	~115	Високий	Середньо-висока	Канонічна GoF-реалізація
C#	~105	Середній	Середня	Менше шаблонного коду
Python	~80	Низький	Низька	Найкомпактніша реалізація

Таблиця 6

Мова	Вплив на патерн
C++	забезпечує строгий контроль структури
Java	патерн формалізує складні конструктори
C#	часто комбінується з fluent-інтерфейсами
Python	має більше концептуальний характер

Для проведення коректного порівняльного аналізу патерна *Prototype* використано ідентичну концептуальну модель: визначено інтерфейс Prototype з методом clone(); реалізовано ConcretePrototype з двома полями стану (FieldA, FieldB); клонування виконується шляхом створення нового об’єкта того ж класу з копіюванням значень полів; використано Prototype Registry, який зберігає прототипи та повертає їх клони за ключем; клієнтський код працює виключно через механізм клонування. Результати порівняння наведені у таблицях 7 та 8.

Таблиця 7

Загальне порівняння

Мова	LOC (≈)	Boilerplate code	Складність написання	Коментар
C++	~110	Високий	Висока	Повний контроль над пам’яттю
Java	~100	Високий	Середня	Відмова від Cloneable підвищує безпеку
C#	~90	Середній	Низько-середня	Проста та чиста реалізація
Python	~70	Низький	Низька	Найкоротший та найчитабельніший код

Таблиця 8

Мова	Вплив на патерн
C++	Prototype необхідний для безпечного копіювання
Java	Патерн замінює небезпечний Cloneable
C#	Чітка, безпечна модель клонування
Python	Prototype часто замінюється copy

Для проведення коректного порівняльного аналізу патерна *Singleton* використано однакову логічну модель побудови: приватне статичне поле, яке зберігає єдиний екземпляр класу; приватний конструктор, що унеможливлює створення об'єкта ззовні; публічний статичний метод доступу, який перевіряє, чи існує екземпляр та за потреби створює його; повертає посилання на нього. Результати порівняння наведені у таблицях 9 та 10.

Таблиця 9

Загальне порівняння

Мова	LOC (≈)	Boilerplate code	Складність написання	Коментар
C++	45	Високий	Середня	Потребує ручного керування пам'яттю
Java	40	Середній	Середня	Класичний GoF-підхід
C#	38	Низький	Низька	Найбільш компактна статично типізована версія
Python	30	Мінімальний	Мінімальна	Логічна імітація приватності

Таблиця 10

Мова	Вплив на патерн
C++	Повний контроль над пам'яттю
Java	Явне розділення відповідальностей.
C#	Реалізація не є потокобезпечною
Python	Відсутність суворих обмежень на створення об'єкта;

Патерн Singleton у всіх реалізаціях повністю відповідає класичному визначенню та працює ідентично з точки зору клієнтського коду. Проте вибір мови програмування суттєво впливає на: обсяг коду; складність реалізації; контроль над ресурсами; продуктивність виконання.

C# є найкращим компромісом між читабельністю, компактністю та строгістю. C++ доцільний у системах з критичними вимогами до швидкодії. Python оптимальний для швидкого прототипування. Java забезпечує класичну, надійну та формалізовану реалізацію.

Висновки. За результатами проведеного аналізу встановлено, що мова *Python* забезпечує найменший обсяг коду та найнижчий поріг входження для реалізації поро-

ISSN 1562-9945 (Print)
ISSN 2707-7977 (Online)

джувальних патернів, що робить її зручною для прототипування та швидкої розробки. Водночас динамічна типізація зменшує рівень формального контролю та може ускладнювати підтримку великих систем.

Мова *C#* продемонструвала оптимальний баланс між строгістю архітектури та зручністю реалізації. Мова дозволяє реалізовувати породжувальні патерни з меншими накладними витратами порівняно з Java, зберігаючи при цьому переваги статичної типізації та сучасних мовних конструкцій.

Мова *Java* забезпечує найбільш формалізований та канонічний підхід до реалізації породжувальних патернів, що робить її зручною для великих корпоративних систем, де особливе значення мають підтримка, масштабованість та чітке дотримання архітектурних стандартів. Разом із тим це супроводжується більшим обсягом коду та значною кількістю шаблонних конструкцій.

Мова *C++*, у свою чергу, надає найвищий рівень контролю над ресурсами та потенційно найкращу продуктивність, однак реалізація породжувальних патернів цією мовою є найбільш складною з точки зору синтаксису, керування пам'яттю та загальним когнітивним навантаженням на розробника.

Загалом результати дослідження підтверджують, що ефективність застосування породжувальних патернів значною мірою залежить не лише від самого патерна, а й від обраної мови програмування та контексту використання. Патерни проєктування не є універсальним рішенням, однак за умови обґрунтованого застосування вони суттєво підвищують якість архітектури програмного забезпечення, спрощують його розвиток та супровід.

ЛІТЕРАТУРА/REFERENCES

1. Gamma E., Helm R., Johnson R., Vlissides J. Design Patterns: Elements of Reusable Object-Oriented Software. - Boston: Addison-Wesley, 1994.

2. Martin R. C. Clean Architecture: A Craftsman's Guide to Software Structure and Design. - Boston: Pearson Education, 2017.

3. Oracle Corporation. The Java Tutorials: Object-Oriented Design Concepts. - Official documentation Java.

4. Microsoft Corporation. Design Patterns in .NET. - Official documentation Microsoft Learn.

5. Gamma E. Design Patterns 25 Years Later: An Interview and Retrospective. - ACM Queue, 2019.

6. Microsoft Learn. Implementing Design Patterns in C#. - Microsoft Learn Documentation. URL: https://learn.microsoft.com/en-us/dotnet/architecture/modern-web-apps-azure/common-web-application-architectures

7. Refactoring.Guru. Creational Design Patterns. - Online Software Design Patterns Guide. URL: https://refactoring.guru/design-patterns/creational-patterns

Received 12.01.2026.
Accepted 16.01.2026.

Comparative analysis of the effectiveness of pattern application

The paper presents a comprehensive theoretical and practical study of creational design patterns. The main attention was focused on the analysis of the mechanisms of object creation, reducing the coupling between system components and increasing the level of abstraction of architectural solutions. Five key patterns were analyzed in detail: Factory Method, Abstract Factory, Builder, Prototype and Singleton. The implementation of the patterns was performed in the programming languages C#, C++, Python and Java. To ensure the correctness and objectivity of the analysis, all patterns were implemented with the same logical structure, which allowed for direct comparison of the obtained solutions. The implementations accounted for the specific characteristics of each programming language, including typing models, syntactic constructs, memory management mechanisms and standard libraries.

For comparative analysis, the implementations were evaluated according to the following criteria:

• number of lines of code (LOC);

• complexity of writing and amount of template code;

• typing and its impact on architectural rigor;

• ease of implementation and readability of the code;

• potential performance and features of compilation or launch.

According to the results of the analysis, Python provides the smallest amount of code and the lowest threshold of entry for implementing generative patterns, which makes it convenient for prototyping and rapid development. At the same time, dynamic typing reduces the level of formal control and can complicate the support of large systems.

C# demonstrated the optimal balance between architectural rigor and ease of implementation. The language allows implementing generative patterns with lower overhead compared to Java, while maintaining the advantages of static typing and modern language constructs.

Java provides the most formalized and canonical approach to implementing generative patterns, making it suitable for large enterprise systems where maintainability, scalability, and strict adherence to architectural standards are of particular importance. However, this comes with a larger amount of code and a significant number of template constructs.

C++, in turn, provides the highest level of control over resources and potentially the best performance, however, the implementation of generative patterns in this language is the most complex in terms of syntax, memory management, and overall cognitive load on the developer.

The conclusions and practical results obtained in the work can be used in object-oriented programming, as well as in the practical activities of software developers when choosing architectural solutions for real software projects.

Матвєєва Наталія Олександрівна – к.т.н., доцент, доцент кафедри електронних обчислювальних машин, Дніпровський національний університет ім. Олеся Гончара.
ORCID: https://orcid.org/0009-0004-3774-5679

Matveeva Nataliya Oleksandrivna – Candidate of Technical Sciences, associate professor, associate professor of the department of Electronic Computing Machinery, Oles Honchar Dnipro National University.
ORCID: https://orcid.org/0009-0004-3774-5679

ISSN 1562-9945 (Print)
ISSN 2707-7977 (Online)