УДК 004.434 https://doi.org/10.32515/2664-262X.2025.11(42).1.37-45

І. А. Котов, проф., д-р техн. наук, **Д. В. Швець**, доц., канд. техн. наук, **Н. О. Карабут**
Криворізький національний університет, м. Кривий Ріг, Україна
e-mail: dmitriy.shvets@knu.edu.ua

Аналіз еволюції об'єктно-орієнтованої парадигми в патернах мови Java для мультиплатформних середовищ

Дослідження спрямоване на аналіз еволюції об’єктно-орієнтованої парадигми у контексті розвитку патернів мови Java для мультиплатформних середовищ. Акцент робиться на тому, як зміни в архітектурних підходах, зумовлені вимогами кроссплатформності, вплинули на трансформацію об’єктно-орієнтованої парадигми — від класичних принципів інкапсуляції та поліморфізму до сучасних гібридних рішень. Мета роботи — виявити зв’язок між адаптацією Java-інструментів та зростанням актуальності патернів, що забезпечують сумісність коду з різноманітними платформами.

У статті простежено еволюційний шлях об’єктно-орієнтованої парадигми у Java, починаючи з ери WORA, де віртуальна машина (JVM) була основним механізмом абстракції від платформ, до сучасних підходів, що поєднують нативну компіляцію з гнучкими архітектурними патернами. На прикладах JavaFX та Spring Boot показано, як інкапсуляція платформо-залежних деталей і поліморфізм перетворилися на інструменти створення універсальних інтерфейсів, здатних адаптуватися під мобільні, десктопні та хмарні середовища.

Детально досліджено роль патернів проектування у контексті мультиплатформності. Вони розглядаються як динамічні механізми, що еволюціонували разом із Java. Окремо аналізуються виклики, такі як зростання складності архітектури при інтеграції з нативними API або обмеження продуктивності JVM у порівнянні зі скомпільованими рішеннями. Показано, як модульність допомагає подолати ці обмеження, зберігаючи переваги об’єктно-орієнтованої парадигми.

Аналіз еволюції об’єктно-орієнтованої парадигми в Java свідчить, що її принципи — інкапсуляція платформо-залежної логіки та поліморфізм для єдиного інтерфейсу — залишаються основою для створення гнучких мультиплатформних систем. JVM поступово доповнюється інструментами, що відкривають доступ до хмарних середовищ, IoT і нативної оптимізації. Патерни проектування трансформуються з класичних шаблонів у механізми адаптації, балансуючи між універсальністю та специфікою платформ. Сучасні виклики, зокрема компроміс між продуктивністю JVM та нативними рішеннями, а також інтеграція з вузькоспеціалізованими API, вимагають глибшого синтезу об’єктно-орієнтованої парадигми з новими підходами. Розвиток гібридних парадигм і фреймворків формує майбутнє Java, де архітектурна гнучкість дозволяє здолати технологічні обмеження, зберігаючи актуальність мови в умовах фрагментації цифрових екосистем.

абстракція, адаптація, інтерфейс, інфраструктура, клас, мультиплатформність, парадигма, патерн, додаток, фреймворк

Постановка проблеми. Сучасна цифрова екосистема характеризується безпрецедентним розмаїттям пристроїв і платформ, що включають мейнфрейми, персональні комп'ютери, мобільні гаджети, IoT-пристрої та хмарні середовища. Це розмаїття зумовлює необхідність створення програмного забезпечення (ПЗ), здатного ефективно працювати в мультиплатформних середовищах без втрати продуктивності та функціональності. Сьогодні мультиплатформність перестала бути опціональною і є обов'язковою вимогою для більшості комерційних і наукових програмних проєктів.

Забезпечення мультиплатформності - це не тільки суто технічне завдання, але також і необхідність переосмислення традиційних підходів до проєктування інформаційних систем, де універсальність не повинна суперечити оптимізації, а гнучкість - ускладнювати код.

Остання обставина призводить до необхідності аналізу розвитку сучасних парадигм інженерії програмного забезпечення. Як базову парадигму розглядають об'єктно-орієнтований підхід (ООП), що реалізується в ефективних патернах для практичних реалізацій у мультиплатформних середовищах. Найбільший інтерес для такого аналізу мають яскраво виражені ООП-орієнтовані мови розробки ПЗ. Безперечно, до таких належить мова програмування Java.

Java на практиці реалізує компроміс між суворою об'єктною типізацією, з одного боку, і переносимістю з можливостями масштабування, з іншого боку. Цей компроміс стає особливо критичним в епоху, коли додатки дедалі частіше реалізуються одночасно на кількох платформах, а їхній життєвий цикл вимірюється роками з вимогою множинних оновлень і розширень.

Виходячи зі сказаного, можна констатувати, що, з урахуванням практично лідируючих позицій, для інфраструктури Java найактуальнішою на сьогодні є проблема забезпечення мультиплатформності реалізації програмних проєктів на ідеологічній основі патернів мови. Останнє забезпечує архітектурні рішення, що дають змогу зберігати сутність додатка незмінною, лише адаптуючи його «оболонку» до вимог конкретного операційного середовища.

Аналіз останніх досліджень і публікацій. Історично Java стала однією з перших мов, які запропонували розв'язання проблеми мультиплатформності через концепцію WORA («Write Once, Run Anywhere») [1]. Віртуальна машина (JVM) абстрагувала розробника від особливостей конкретних операційних систем. Однак із часом вимоги до мультиплатформності ускладнилися: сьогодні недостатньо просто запустити код на різних ОС - потрібно забезпечити адаптацію інтерфейсів, оптимізацію під апаратні особливості та інтеграцію з екосистемами конкретних пристроїв (наприклад, мобільні push-повідомлення або десктоп-інтеграції).

У цьому контексті ООП виступає не просто парадигмою, а стратегічним інструментом для створення гнучких масштабованих архітектур. Принципи ООП, такі як інкапсуляція та поліморфізм, дають змогу ізолювати платформо-залежну логіку, зберігаючи ядро програмної системи незалежним від зовнішніх умов. Наприклад, абстракція графічного інтерфейсу в JavaFX [2] або Swing [3] дає можливість розробляти єдиний UI-шар, який адаптується під різні оточення, зберігаючи узгодженість внутрішньої бізнес-логіки.

Очевидною є тенденція - нативна розробка сьогодні активно еволюціонує в напрямі кросплатформного підходу, що активно розвивається розробниками та досліджується науковцями [4, 5]. Незважаючи на те, що платформу Java було презентовано ще наприкінці XX століття, сьогодні вона отримує свій розвиток у контексті мультиплатформної розробки мобільних застосунків [6, 7], зокрема з використанням мови програмування Kotlin [8], яка працює на базі JVM [9, 10].

Постановка завдання. У роботі поставлено завдання аналізу впливу парадигми ООП на мультиплатформну адаптацію коду шляхом впровадження патернів у практику проєктування для забезпечення гнучкості та масштабованості кросплатформних систем, а також виявлення технічних та архітектурних складнощів, що виникають під час розроблення універсальних рішень на Java.

Викладення основного матеріалу. ООП сьогодні можна розглядати як філософію проєктування, що трансформує складність мультиплатформних систем у керовану структуру. Його перевага полягає у здатності створювати абстракції, які відображають сутність предметної області, залишаючись інваріантними щодо зовнішніх умов. У контексті мультиплатформності ООП дає змогу відокремити абстрактну сутнісну функціональність від практичної реалізації в конкретному середовищі, що забезпечує можливість порівняно легкої адаптації до платформи.

Адаптивність реалізується через багаторівневу архітектуру, де кожен рівень інкапсулює певний аспект роботи застосунку. Взаємодія з операційною системою або

апаратними ресурсами може бути прихована за інтерфейсами, які залишаються незмінними для бізнес-логіки. Цей механізм адаптує універсальні інструкції в зрозумілі конкретній платформі команди, не вимагаючи коригування основного коду Java з її акцентом на інтерфейси й абстрактні класи та надаючи природне середовище для вироблення рішень - чи то робота з файловою системою через *java.nio* [11], де прихованими залишаються відмінності між Windows і Linux, чи то рендеринг графіки в JavaFX, що абстрагується від нативних GUI-бібліотек.

Важливим фактором адаптивності є поліморфізм, що дає змогу програмним об'єктам набувати різних форм залежно від контексту виконання. Наприклад, клас, що відповідає за мережеву комунікацію, може мати реалізації для десктопних застосунків (на базі стандартних сокетів) і мобільних пристроїв (з оптимізацією під енергоефективність), але викликатися через єдиний метод *sendData()*. Це не тільки спрощує підтримку коду, а й забезпечує можливість «підключати» в міру необхідності платформо-залежні модулі.

Водночас адаптивність означає здатність системи еволюціонувати. Спадкування і композиція в ООП дають змогу розширювати функціональність без порушення наявної логіки. Наприклад, під час додавання підтримки нової IoT-платформи розробник може створити спадкоємця базового класу *DeviceController*, перевизначивши методи ініціалізації та обміну даними, зберігаючи водночас загальну структуру управління пристроями. У Java це особливо важливо з огляду на її застосування в «довгоживучих» масштабованих enterprise-системах, де вимоги до підтримки нових платформ виникають регулярно. Однак необхідно зазначити, що надлишкова абстракція може призвести до ситуації, коли код стає занадто загальним, втрачаючи ефективність на конкретних платформах. Наприклад, спроба створити універсальний UI-компонент для всіх пристроїв іноді обертається втратою нативної «чутливості» компонента. І тут на допомогу приходять патерни проєктування. Вони нормалізують баланс між універсальністю та специфічністю. У Java це проявляється в підходах, коли фреймворки (наприклад, Spring Boot [12]) використовують dependency injection [13] для динамічного вибору реалізацій залежно від оточення, зберігаючи основу програми платформо-незалежною.

Таким чином, можна сформувати загальну парадигму для підходів до кросплатформного розроблення програмного забезпечення, яка водночас спирається на принципи ООП, платформо-незалежну інтерпретацію та абстракції, які реалізуються через патерни. Цю схему наведено на рисунку 1.

Мова Java реалізує зазначені вище принципи. Позитивною особливістю Java є її здатність стирати кордони між платформами, перетворюючи код на універсальну мову, зрозумілу будь-якому середовищу. JVM не просто інтерпретує байт-код, а створює ізольоване середовище, де застосунок функціонує за єдиними правилами, незалежно від того, чи запущено його на Windows, Linux, чи вбудовано в мікроконтролер через Java ME.

Однак сьогодні підтримка принципів мультиплатформності вимагає більшого, ніж проста сумісність на *runtime*-рівні. Розробникам необхідно створювати інтерфейси, які виглядають «нативно» на кожному пристрої, інтегруватися з хмарними сервісами та керувати ресурсами в умовах обмежень мобільних гаджетів. Вирішення цих завдань забезпечують фреймворки, перетворюючи абстрактні ідеї WORA на конкретні інструменти. Наприклад, JavaFX пропонує єдину модель для побудови графічних інтерфейсів, які автоматично адаптуються під стиль macOS, Windows або мобільних ОС, зберігаючи логіку взаємодії незмінною. А вищезгаданий Spring Boot, що став стандартом для enterprise-рішень, дає змогу розгортати додатки в будь-якому середовищі - від локального сервера до хмарного кластера, абстрагуючись від специфіки інфраструктури.

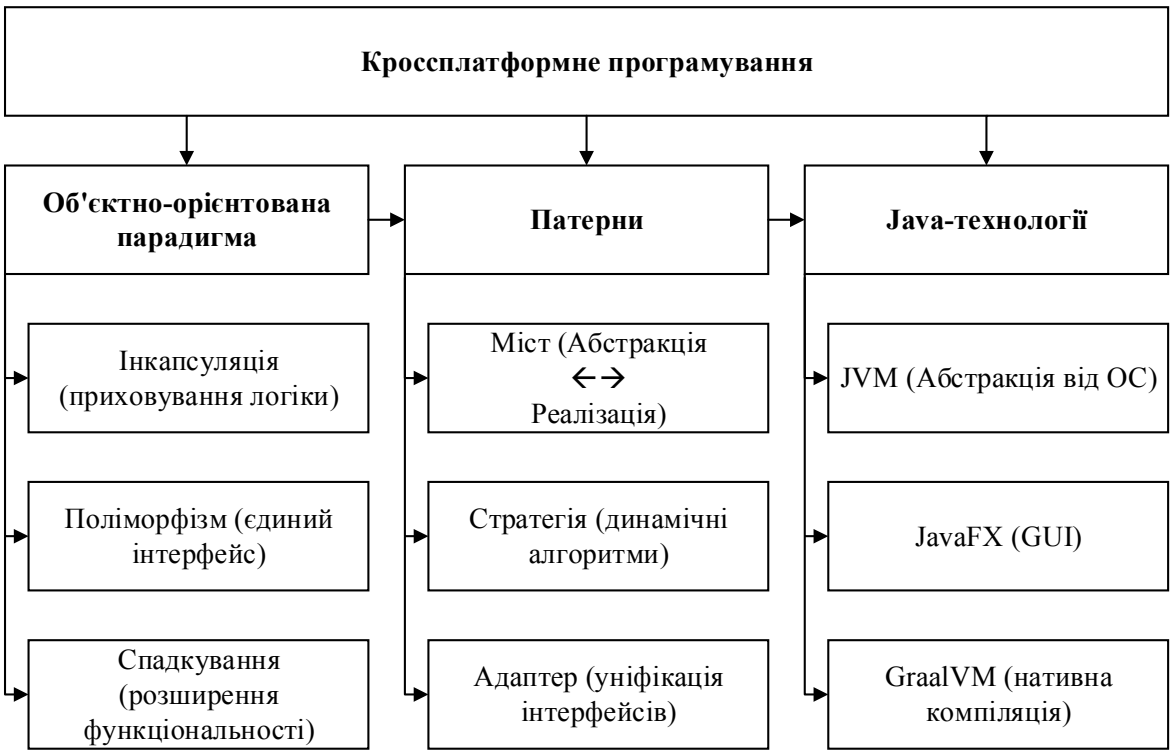

Рисунок 1 – Взаємозв'язок складових кросплатформного програмування
Джерело: розроблено авторами

Важливим кроком в еволюції Java стала модульність, яка перетворила монолітні додатки на набори гнучких компонентів. Наприклад, модуль для роботи з сенсорами може бути активований тільки в мобільній збірці, а десктоп-версія отримає розширені можливості роботи з файлами. Така вибірковість знижує накладні витрати і робить додатки більш адаптивними до вимог цільового середовища.

Водночас Java не обмежується «віртуальним» світом. Такі проекти, як GraalVM [14], розширюють її можливості, компілюючи байт-код у нативні бінарники для Linux, macOS або навіть IoT-пристроїв. Це значною мірою знімає взаємні протиріччя між кросплатформністю та нативною продуктивністю, даючи змогу Java конкурувати з мовами на кшталт C++ у галузях, де раніше її вважали непридатною. Інструменти типу RoboVM демонструють, як Java-код може взаємодіяти з iOS-бібліотеками, перетворюючись на «міст» між екосистемами. Бібліотеки на кшталт Apache Commons [15] або Google Guava [16] пропонують готові абстракції для роботи з мережею, файлами або багатопоточністю, приховуючи платформо-залежні деталі за універсальними інтерфейсами. Навіть такі «дрібниці», як опрацювання переведення рядків у різних ОС, вирішуються через класи та відповідні методи (наприклад, *System.lineSeparator()*), які автоматично підлаштовуються під середовище виконання.

Детальний аналіз патернів проектування в Java демонструє, що їх можна розглядати як деяку мову, якою архітектура веде діалог з мінливістю платформ. Патерни формалізували досвід поколінь програмістів, перетворивши хаос платформо-залежних нюансів на систематизовані впорядковані структури. Загальну класифікацію патернів проілюстровано на рисунку 2.

У контексті мультиплатформності саме патерни стають мостами між вимогами абстракції системи та конкретними реалізаціями, даючи змогу коду адаптуватися до оточення, зберігаючи цілісність задуму розробника. У такий спосіб реалізується ідея

розділення абстракції та реалізації, що дає змогу додатку залишатися собою, змінюючи лише зовнішній вигляд під вимоги платформи. Саме так працює патерн *Bridge*, де інтерфейс для роботи з графікою може мати одну абстракцію для візуалізації кнопок, але різні реалізації під Windows, macOS або мобільний WebView. Цей ефект досягається в Java, коли фреймворки на кшталт libGDX використовують єдиний API для рендерингу, делегуючи низькорівневі виклики OpenGL (на ПК) або GLES (на Android), не зачіпаючи ігрову логіку [18].

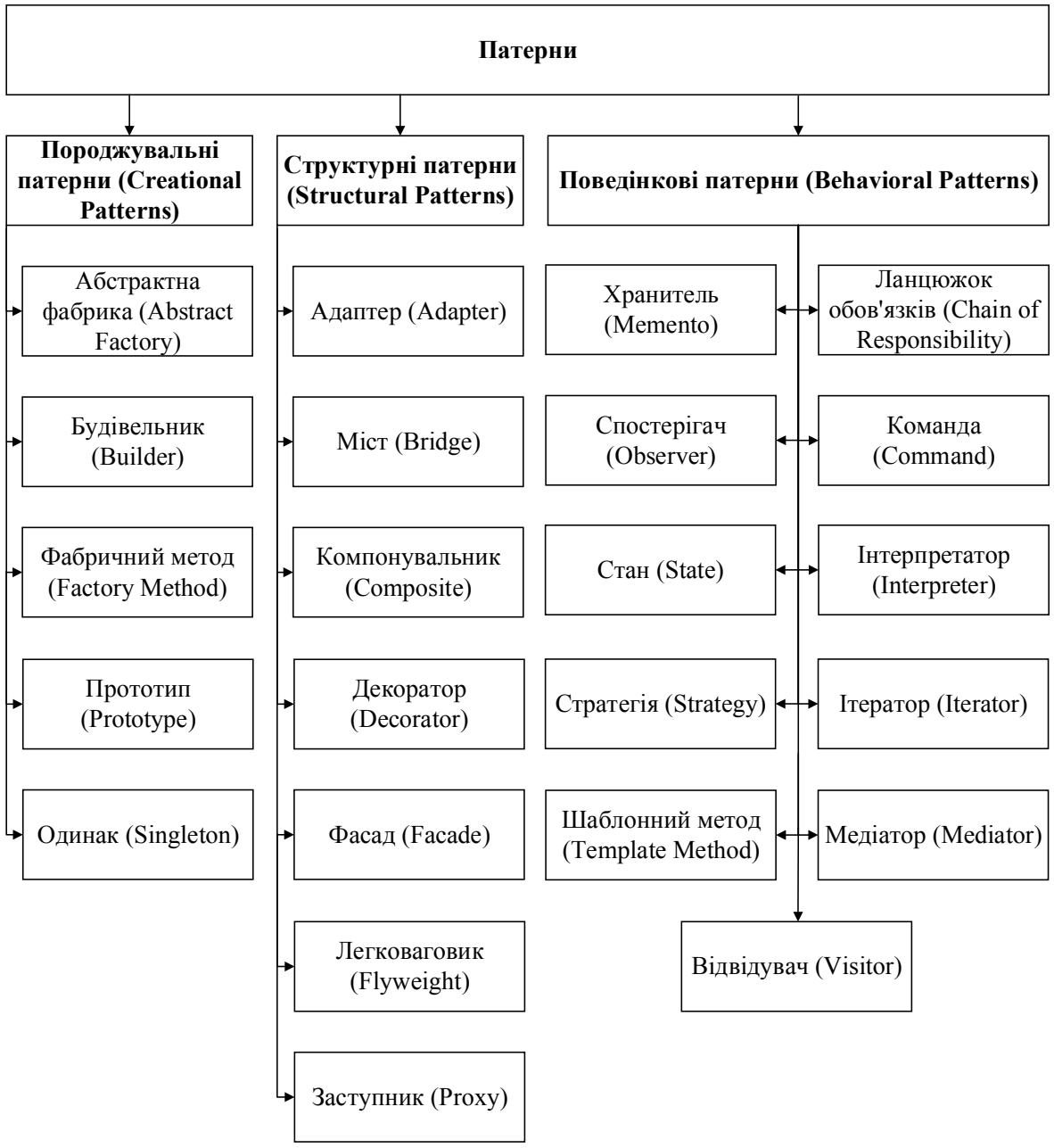

Рисунок 2 – Типи та різновиди патернів

Джерело: розроблено на основі [17]

Іншим важливим аспектом реалізації патернів є динамічна поведінка. Патерн *Strategy* упаковує платформо-специфічні алгоритми у взаємозамінні модулі, які можна «вбудовувати» безпосередньо в *runtime*-середовище. Наведемо приклад. Мережева

взаємодія може використовувати різні протоколи шифрування залежно від обмежень платформи: TLS 1.3 для десктопа і спрощений алгоритм для IoT-пристроїв з обмеженими ресурсами. Усе це приховується єдиним методом *sendSecureData()*, а вибір стратегії здійснюється через конфігурацію або автоматичне визначення середовища виконання. Очевидно, що в Java це особливо зручно, завдяки інтерфейсам і впровадженню залежностей, які дають змогу «збирати» застосунок із рішень, що підходять для конкретного контексту.

Іноді під час мультиплатформної адаптації потрібна трансформація даних між контекстами додатків. Для вирішення цього завдання використовується патерн *Adapter* як універсальний перекладач. Наприклад, мобільний застосунок може отримувати геолокаційні дані через Google Maps API на Android і CoreLocation на iOS, але представляти їх у єдиному форматі для внутрішньої логіки. У Java подібне реалізується через обгортки над нативними бібліотеками, де *Adapter* приховує відмінності в JSON-структурах або системних викликах, перетворюючи їх на стандартизовані об'єкти.

Сила патернів полягає у здатності створювати екосистеми абстракцій. Наприклад, патерн *Factory* може генерувати екземпляри класів, специфічних для платформи, але робить це через єдину точку входу. Як приклад можна навести сервіс для роботи з повідомленнями: на десктопі він використовує системний трей, на мобільному пристрої - *Push*-сервіси, а у веб-версії - браузерні API. *Factory*, аналізуючи оточення, створює потрібну реалізацію, але для решти коду всі вони виглядають як єдиний інтерфейс *NotificationService*. У Java такі методи *Factory* часто поєднуються з *Reflection* або *ServiceLoader*, щоб динамічно підвантажувати реалізації без жорстких залежностей.

Як приклад наведемо фрагмент короткого лістингу використання патерну *Factory Method Pattern* для побудови класів Java для парсингу різних XML-документів [17]:

```
package com.javacodegeeks.patterns.factorymethodpattern;
	public interface XMLParser {
		public String parse(); }

public class ErrorXMLParser implements XMLParser {
	@Override
	public String parse() {
		System.out.println("Parsing error XML...");
		return "Error XML Message"; }
}

public class FeedbackXML implements XMLParser {
	@Override
	public String parse() {
		System.out.println("Parsing feedback XML...");
		return "Feedback XML Message"; }
}

public class OrderXMLParser implements XMLParser {
	@Override
	public String parse() {
		System.out.println("Parsing order XML...");
		return "Order XML Message"; }
}

public class ResponseXMLParser implements XMLParser {
	@Override
	public String parse() {
		System.out.println("Parsing response XML...");
		return "Response XML Message"; }
}
```

```
public abstract class DisplayService {
	public void display(){
		XMLParser parser = getParser();
		String msg = parser.parse();
		System.out.println(msg); }

protected abstract XMLParser getParser();
}

public class ErrorXMLDisplayService extends DisplayService {
	@Override
	public XMLParser getParser() {
		return new ErrorXMLParser(); }
}

public class FeedbackXMLDisplayService extends DisplayService {
	@Override
	public XMLParser getParser() {
		return new FeedbackXML(); }
}

public class OrderXMLDisplayService extends DisplayService {
	@Override
	public XMLParser getParser() {
		return new OrderXMLParser(); }
}

public class ResponseXMLDisplayService extends DisplayService {
	@Override
	public XMLParser getParser() {
		return new ResponseXMLParser(); }
}
```

Розробнику необхідно уникати ілюзії універсальності, коли абстракції стають занадто «товстими», втрачаючи зв'язок із реальністю платформ. Патерн *Abstract Factory* вирішує цю проблему, групуючи логічно пов'язані компоненти. Наприклад, створення UI-елементів для різних платформ вимагає узгоджених стилів: кнопка, чекбокс і меню повинні виглядати як частина однієї екосистеми. Фабрика для Android генерує *Material Design*-компоненти, а для iOS - *Cupertino*-стиль, але гарантує, що всі елементи інтерфейсу поєднуватимуться між собою. У JavaFX це реалізується через *skins* і CSS-стилізацію, де *Factory* керує застосуванням тем залежно від ОС.

Висновки. Аналіз еволюції об'єктно-орієнтованої парадигми в патернах мови Java показав, що принципи ООП (інкапсуляція платформо-залежної логіки та поліморфізм для єдиного інтерфейсу) є ключовими для створення гнучких мультиплатформних програмних систем. Вони дають змогу розділити абстракцію та реалізацію, мінімізуючи дублювання коду. При цьому JVM залишається фундаментом кросплатформності Java. Сучасні концепції та інструменти додатково розширюють її можливості, забезпечуючи інтеграцію з хмарними середовищами, IoT і нативними платформами. Зі свого боку, модульність і компіляція в нативний код відкривають нові можливості для подальшої оптимізації.

Використання патернів формалізує підходи до мультиплатформної адаптації, перетворюючи різнорідні реалізації на узгоджені модулі. Це дає можливість балансувати між універсальністю та платформо-специфічною оптимізацією, знижуючи вартість підтримки.

Незважаючи на перераховані позитивні ефекти, кросплатформна і патерно-орієнтована розробка на Java стикається з певними проблемами компромісу між продуктивністю JVM і нативною швидкістю, складністю інтеграції з унікальними API

мобільних і IoT-пристроїв, ризиком надлишкової абстракції, яка призводить до ускладнення архітектури. Але, безсумнівно, розвиток гібридних парадигм і фреймворків забезпечує подальшу еволюцію і стійке зростання ефективності кросплатформного підходу Java.

Список літератури

1. What Gives Java its Write Once Run Anywhere Nature? *Itvedant.com.* URL: https://www.itvedant.com/blog/java-write-once-run-anywhere (date of access: 10.02.2025).
2. Chin S., Vos J., Weaver J. The Definitive Guide to Modern Java Clients with JavaFX. Berkeley, CA : Apress, 2024. 626 c. DOI: 10.1007/979-8-8688-0998-9.
3. David J. Eck. Introduction to Programming Using Java, Version 9, Swing Edition Series. New York : Hobart and William Smith Colleges, 2022.
4. Pinto C. M., Coutinho C. From Native to Cross-platform Hybrid Development. *2018 International Conference on Intelligent Systems (IS)*, м. Funchal - Madeira, Portugal, 25–27 верес. 2018 р. 2018. DOI: 10.1109/is.2018.8710545.
5. Comparative analysis of cross-platform development methodologies: a comprehensive study / R. Jangassiyev та ін. *TELKOMNIKA (Telecommunication Computing Electronics and Control)*. 2024. Т. 23, № 1. С. 108. DOI: 10.12928/telkomnika.v23i1.26331.
6. Hiwale P. R. Review On Cross-Platform Mobile Application Development. *International Journal for Research in Applied Science and Engineering Technology*. 2022. Т. 10, № 1. С. 1433–1439. DOI: 10.22214/ijraset.2022.40004.
7. You D., Hu M. A Comparative Study of Cross-platform Mobile Application Development. *12th Annual Conference of Computing and Information Technology Research and Education*, New Zealand. 2021.
8. Evaluating Kotlin Multiplatform: Superior cross-platform development / A. Punia та ін. *SSRN Electronic Journal.* 2024. DOI: 10.2139/ssrn.4836587.
9. Performance Study of Kotlin and Java Program Considering Bytecode Instructions and JVM JIT Compiler / A. Sonoyama та ін. *2021 Ninth International Symposium on Computing and Networking Workshops (CANDARW)*, м. Matsue, Japan, 23–26 листоп. 2021 р. 2021. DOI: 10.1109/candarw53999.2021.00028.
10. Nagy R. Simplifying Application Development with Kotlin Multiplatform Mobile: Write robust native applications for iOS and Android efficiently. Birmingham : Packt Publishing, 2022. 184 с.
11. Java NIO. *Oracle.com*. URL: https://docs.oracle.com/en/java/javase/22/core/java-nio.html (дата звернення: 12.02.2025).
12. Spring Boot. *Spring.io*. URL: https://spring.io/projects/spring-boot (дата звернення: 12.02.2025).
13. Java Dependency Injection. *Digitalocean.com.* URL: https://www.digitalocean.com/community/tutorials/java-dependency-injection-design-pattern-example-tutorial (дата звернення: 14.02.2025).
14. GraalVM. *Graalvm.org*. URL: https://www.graalvm.org/ (дата звернення: 14.02.2025).
15. Apache Commons. *Commons.apache.org*. URL: commons.apache.org (дата звернення: 15.02.2025).
16. Google Guava. *Github.com*. URL: https://github.com/google/guava (дата звернення: 15.02.2025).
17. Joshi R. Java Design Patterns. Kesariani : Exelixis Media, 2015. 173 с.
18. libGDX. *Libgdx.com.* URL: https://libgdx.com/ (дата звернення: 16.02.2025).

References

1. Itvedant.com. (n.d.). *What Gives Java its Write Once Run Anywhere Nature?* Retrieved February 10, 2025, from https://www.itvedant.com/blog/java-write-once-run-anywhere
2. Chin, S., Vos, J., & Weaver, J. (2024). *The Definitive Guide to Modern Java Clients with JavaFX*. Berkeley, CA: Apress. doi:10.1007/979-8-8688-0998-9
3. Eck, D. J. (2022). *Introduction to Programming Using Java, Version 9, Swing Edition Series*. New York: Hobart and William Smith Colleges.
4. Pinto, C. M., & Coutinho, C. (2018). From Native to Cross-platform Hybrid Development. In *2018 International Conference on Intelligent Systems (IS)*. IEEE. doi:10.1109/is.2018.8710545
5. Jangassiyev, R., Umarova, Z., Ussenova, A., Makhanova, Z., Zhumatayev, N., Amirov, M., & Koishibekova, G. (2024). Comparative analysis of cross-platform development methodologies: a comprehensive study. *TELKOMNIKA (Telecommunication Computing Electronics and Control), 23*(1), 108. doi:10.12928/telkomnika.v23i1.26331.
6. Hiwale, P. R. (2022). Review On Cross-Platform Mobile Application Development. *International Journal for Research in Applied Science and Engineering Technology, 10*(1), 1433–1439. doi:10.22214/ijraset.2022.40004.

7. You, D., & Hu, M. (2021). A Comparative Study of Cross-platform Mobile Application Development. In *12th Annual Conference of Computing and Information Technology Research and Education.*
8. Punia, A., Singh, A., Goyal, A., & Arya, A. (2024). Evaluating Kotlin Multiplatform: Superior cross-platform development. *SSRN Electronic Journal*. doi:10.2139/ssrn.4836587
9. Sonoyama, A., Kamiyama, T., Oguchi, M., & Yamaguchi, S. (2021). Performance Study of Kotlin and Java Program Considering Bytecode Instructions and JVM JIT Compiler. In *2021 Ninth International Symposium on Computing and Networking Workshops (CANDARW)*. IEEE. doi:10.1109/candarw53999.2021.00028
10. Nagy, R. (2022). *Simplifying Application Development with Kotlin Multiplatform Mobile: Write robust native applications for iOS and Android efficiently.* Birmingham: Packt Publishing.
11. Oracle. (n.d.). *Java NIO*. Retrieved February 12, 2025, from docs.oracle.com/en/java/javase/22/core/java-nio.html
12. Spring Boot. (n.d.). Retrieved February 12, 2025, from https://spring.io/projects/spring-boot
13. Java Dependency Injection. (n.d.). *DigitalOcean*. Retrieved February 14, 2025, from www.digitalocean.com/community/tutorials/java-dependency-injection-design-pattern-example-tutorial
14. GraalVM. (n.d.). Retrieved February 14, 2025, from https://www.graalvm.org/
15. Apache Commons. (n.d.). Retrieved February 15, 2025, from https://commons.apache.org/
16. Google. (n.d.). *Guava*. GitHub. Retrieved February 15, 2025, from https://github.com/google/guava
17. Joshi, R. (2015). *Java Design Patterns*. Exelixis Media.
18. libGDX. (n.d.). Retrieved February 16, 2025, from https://libgdx.com/.

Ihor Kotov, Prof., DSc., **Dmitriy Shvets,** Assoc. Prof., PhD tech. sci., **Nadia Karabut**
Kryvyi Rih National University, Kryvyi Rih, Ukraine

Analyzing the Evolution of Object-Oriented Paradigm in Java Language Patterns for Multiplatform Environments

Research focused on the evolution of the object-oriented paradigm in the context of Java language pattern development for cross-platform environments. It centralizes on analyzing the evolution of the object-oriented paradigm in the context of developing Java language patterns for cross-platform environments. Emphasis is placed on how changes in architectural approaches, driven by cross-platform requirements, have influenced the transformation of the object-oriented paradigm — from classical principles of encapsulation and polymorphism to modern hybrid solutions. The work aims to identify the connection between the adaptation of Java tools and the increasing relevance of patterns that ensure code compatibility across various platforms.

The paper traces the evolutionary path of the object-oriented paradigm in Java, starting from the WORA era, where the Java Virtual Machine (JVM) served as the primary mechanism for platform abstraction, to modern approaches combining native compilation with flexible architectural patterns. Examples such as JavaFX and Spring Boot demonstrate how the encapsulation of platform-dependent details and polymorphism have evolved into tools for creating universal interfaces that adapt to mobile, desktop, and cloud environments.

The role of design patterns in cross-platform development is examined in detail. Patterns are considered dynamic mechanisms that evolved alongside Java itself. Particular attention is paid to challenges such as increasing architectural complexity when integrating with native APIs or the performance limitations of the JVM compared to compiled solutions. It is shown how modularity helps overcome these limitations while preserving the advantages of the object-oriented paradigm.

An analysis of the evolution of the object-oriented paradigm in Java confirms that its principles — encapsulation of platform-dependent logic and polymorphism for a unified interface — remain the foundation for creating flexible cross-platform systems. The JVM is gradually supplemented with tools providing access to cloud environments, IoT, and native optimization. Design patterns evolve from classical templates into adaptive mechanisms that balance universality and platform specificity. Modern challenges, particularly the trade-off between JVM performance and native solutions and integration with highly specialized APIs, demand a more profound synthesis of object-oriented paradigms with new approaches. The development of hybrid paradigms and frameworks shapes the future of Java, where architectural flexibility allows for overcoming technological constraints while maintaining the language's relevance in a fragmented digital ecosystem.

abstraction, adaptation, interface, infrastructure, class, multiplatform, paradigm, pattern, application, framework

Одержано (Received) 28.02.2025 *Прорецензовано (Reviewed) 07.03.2025*
Прийнято до друку (Approved) 14.03.2025