УДК 004.624 DOI https://doi.org/10.35546/kntu2078-4481.2025.3.2.13

В. В. ВОЙТКО
кандидат технічних наук, доцент,
доцент кафедри програмного забезпечення
Вінницький національний технічний університет
ORCID: 0000-0002-3329-7256

М. Ю. ПОЗУР
аспірант кафедри програмного забезпечення
Вінницький національний технічний університет
ORCID: 0009-0003-5225-2453

МЕТОД РОЗШИРЕННЯ МОЖЛИВОСТЕЙ ПОВТОРНОГО ВИКОРИСТАННЯ ПРОГРАМНИХ КОМПОНЕНТІВ ШЛЯХОМ РЕАЛІЗАЦІЇ TRAITS В .NET

У статті розглядається метод розширення можливостей повторного використання програмних компонентів шляхом реалізації traits в .NET. Проаналізовано основні методи покращення можливостей повторного використання програмних компонентів в об'єктно-орієнтованій парадигмі програмування. Проаналізовано переваги та недоліки множинного та одиничного наслідування. Встановлено, що множинне наслідування має позитивний ефект на модульність та на можливості повторного використання компонентів, проте його використання може призвести до складнішої ієрархії класів та не завжди підтримується мовою програмування. В якості альтернативи наслідуванню розглянуто використання композиції для забезпечення повторного використання програмних компонентів. Проаналізовано концепт traits, який реалізовано у низці сучасних мов програмування в якості альтернативи множинному наслідуванню. В результаті аналізу встановлено, що traits дозволяє отримати схожий до множинного наслідування функціонал. Також проаналізовано пропозиції щодо розширення концепту за рахунок додавання можливості визначати стани, а не тільки поведінку. На основі цього розроблено модель traits для реалізації в .NET, що дозволяє визначати стани та забезпечити кращий контроль над перевантаженням членів при композиції. Оскільки композицію можна реалізувати за рахунок використання інструментів метапрограмування, то було проаналізовано відповідний інструментарій платформи .NET. У результаті аналізу визначено, що інструмент Incremental Source Generators у комбінації з Roslyn API дозволить досягти бажаного результату. Для цього запропоновано використовувати генератор коду, що буде генерувати часткові об'явлення класів, які міститимуть члени відповідного trait'а. В якості альтернативи ключовим словам запропоновано використовувати атрибути метаданих для контролю поведінки такого генератора коду. Генератор коду має спрацьовувати не лише під час компіляції, а й при змінах у вихідному коді. Такий метод дозволить повною мірою реалізувати концепт traits в .NET, що сприятиме покращенню можливостей повторного використання програмних компонентів при розробці на цій платформі.

Ключові слова: *метапрограмування, .NET, генерація коду, розробка програмного забезпечення, traits.*

V. V. VOITKO
Candidate of Technical Sciences, Associate Professor,
Associate Professor at the Department of Software Engineering
Vinnytsia National Technical University
ORCID: 0000-0002-3329-7256

M. YU. POZUR
Postgraduate Student at the Department of Software Engineering
Vinnytsia National Technical University
ORCID: 0009-0003-5225-2453

METHOD OF IMPROVING CODE REUSABILITY BY IMPLEMENTING TRAITS IN .NET

The article describes a method improving code reusability by implementing traits in .NET. The main methods for improving the possibilities of reusing software components in the object-oriented programming paradigm are analyzed. The advantages and disadvantages of multiple and single inheritance are analyzed. It is established that multiple inheritance has a positive effect on modularity and on the code reusability, however, it can lead to a more complex class hierarchy, has more complex semantics and is not always supported by the programming language. As an alternative to inheritance, the use of composition to ensure the reuse of software components is considered. The concept of traits is analyzed, which

utilizes composition and is implemented in a number of modern programming languages as an alternative to multiple inheritance. As a result of the analysis, it is established that traits allow you to obtain functionality similar to multiple inheritance without drawbacks it causes. A number of proposals for expanding the concept by adding the ability to define states are also analyzed. Based on this, a traits model was developed for implementation in .NET, which allows defining states and providing better control over member overriding during composition. Since composition can be implemented by utilizing metaprogramming techniques, the corresponding .NET platform metaprogramming toolkit was analyzed. As a result of the analysis, it was determined that the Incremental Source Generators tool in combination with the Roslyn API will allow achieving the desired result. For this purpose, it is proposed to use a code generator that will generate partial declarations of classes that will contain members of the corresponding trait. As an alternative to keywords, it is proposed to use metadata attributes to control the behavior of such a code generator. Such a code generator should work not only during compilation, but also react to the source code changes. This method will allow fully implementing the traits concept in .NET, which will help improve code reusability when developing on this platform.

Key words: *metaprogramming, .NET, code generation, software development, traits.*

Постановка проблеми

Платформа .NET та мова програмування C# реалізують об'єктно-орієнтовану парадигму програмування. Основним засобом повторного використання коду в такій парадигмі є механізм наслідування. В об'єктно-орієнтованому програмуванні наслідування забезпечує вертикальне повторне використання (vertical reuse), де клас-нащадок може використовувати поведінку та стани, визначені в батьківському класі. Саме ж наслідування може бути одиничним або множинним. Більшість сучасних мов програмування реалізують лише одиничне наслідування. Це обумовлено тим, що множинне наслідування має низку недоліків, що потребують складних механізмів на рівні мови програмування для їх усунення або мінімізації. Найбільш значною є проблема ромбовидного наслідування. Проте множинне наслідування дозволяє забезпечити значно кращі можливості для повторного використання коду в порівнянні з одиничним наслідуванням [1].

Альтернативою наслідуванню є використання композиції. На відміну від наслідування, що дозволяє нащадку використовувати поведінку й стани батьківського класу, композиція дозволяє включати до класу поведінку або стани з іншого класу або спеціально визначеного синтаксичного типу. Це дозволяє забезпечити горизонтальне повторне використання (horizontal reuse). Одним із концептів, що опирається на використання композиції для забезпечення кращих можливостей повторного використання коду, є traits [2]. Концепт передбачає наявність спеціальних синтаксичних одиниць, які називаються trait. Такі синтаксичні одиниці не беруть участі в наслідуванні та можуть визначати як поведінку, так і стани. Traits реалізовано в низці сучасних мов програмування, таких як: Rust, PHP та Scala.

Реалізація концепту traits в .NET та мови програмування C# дозволить значно покращити можливості повторного використання коду при розробці програмного забезпечення на цій платформі [3].

Аналіз останніх досліджень і публікацій

Було проведено аналіз публікацій та розробок, пов'язаних із темою дослідження. В першу чергу, було розглянуто низку публікацій, що описують особливості концепту traits та визначають семантику і модель їх композиції[2]. Оскільки оригінальне визначення концепту передбачає, що trait не може містити стани, то було проаналізовано публікацію, автори якої пропонують розширити концепт за рахунок можливості визначати стани [4].

Так як у статті розглядається реалізація концепту traits шляхом використання засобів метапрограмування платформи .NET, а саме інструментів Roslyn Source Generators [5] та Roslyn API, то було проаналізовано публікації та розробки, в яких описано використання цих інструментів. Наприклад, автори статті [6] описують використання Roslyn API для автоматичної генерації інтеграційних тестів для додатків, написаних мовою програмування C#. У статті [7] описано розробку бібліотеки, що дозволяє спростити написання unit-тестів за допомогою Roslyn API.

Також було проаналізовано бібліотеки, що використовують Roslyn Source Generators для реалізації функціоналу рівня мови програмування [8, 9].

Формулювання мети дослідження

Метою дослідження є розширення можливостей повторного використання коду в .NET шляхом розробки методу з реалізацією концепту traits та використанням інструментів метапрограмування платформи .NET, що дозволить розширити функціонал мови програмування та збільшити кількість сценаріїв повторного використання програмних компонентів.

Викладення основного матеріалу дослідження

Розглянемо оригінальну модель traits, яка описана у статті [10]. Запропонована модель виділяє такі основні множини:

– N – множина імен методів класу,
– B – множина тіл методів класу,
– A – множина змінних, що визначають стан екземпляру класу.

Метод класу визначається як $a \mapsto m$, де $a \in N$, $m \in B$. Множина тіл класів B розширена до B^* за рахунок подання елементів $\mathbb{U}$ та $\mathbb{C}$, що позначають обов'язковий (невизначений) метод та конфлікт методів відповідно.

Таким чином, набір методів класу можна визначити як $d \in D$, $d : N \to B^*$, який містить скінченну множину елементів. Клас $c \in \mathcal{C}$ у такому випадку подається у вигляді $c = \langle \alpha, d \rangle$, де $\alpha \in A$, набір змінних, що визначають стан екземпляру класу.

Автори визначать trait $t \in T$ як $t : N \to B^*$, де $t^{-1}(B \cup \mathbb{C})$ є скінченною множиною. Таким чином, trait – це словник методів, що не може містити станів. Виходячи з цього, модель визначає правила та операції для композиції trait і класу та trait і trait. Це необхідно для забезпечення можливостей вирішення конфліктів імен. Сама ж композиція визначається за виразом (1):

$$(d + t)(l) = d(l) \sqcup t(l). \tag{1}$$

Операція $\sqcup$ для визначень методів B^* виконується за правилами:

$\sqcup$	$\mathbb{U}$	m_1	m_2	$\mathbb{C}$
$\mathbb{U}$	$\mathbb{U}$	m_1	m_2	$\mathbb{C}$
m_1	m_1	m_1	$\mathbb{C}$	$\mathbb{C}$
m_2	m_2	$\mathbb{C}$	m_2	$\mathbb{C}$
$\mathbb{C}$	$\mathbb{C}$	$\mathbb{C}$	$\mathbb{C}$	$\mathbb{C}$

Основним механізмом вирішення конфліктів імен, передбачених у моделі, є перевантаження $d \triangleright t : N \to B^*$ (2):

$$(d \triangleright t)(l) = \begin{cases} t(l) & \text{якщо } d(l) = U \\ d(l) & \end{cases}. \tag{2}$$

Модель також передбачає можливість визначення аліасу для методу.

Це дозволяє визначати додаткове ім'я для методу, яке може бути використане у випадку, коли виникає конфлікт імен. Така операція визначається за правилом (3):

$$d[a \to b](l) = \begin{cases} d(l) & \text{якщо } l \neq a \\ d(b) & \text{якщо } l = a \wedge d(a) = U \\ C & \end{cases}. \tag{3}$$

Для того, щоб типаж міг визначати стани, пропонується розширити існуючу модель. Для цього визначимо основні множини:

- N_m – множина ідентифікаторів методів класу;
- B_m – множина визначень методів класу;
- N_s – множина ідентифікаторів станів класу;
- B_s – множина визначень станів класу.

Аналогічно до існуючої, така модель визначає набір методів класу або trait'а як словник $d \in D$, $d : N_m \to B_m^*$. Стани у цій моделі також визначаються як словник $s \in S$, де $s : N_s \to B_s^*$. Таким чином, клас можна подати як $c = \langle s, d \rangle$. Варто зауважити, що модель розглядається виключно в контексті станів та поведінки, тому інші типи членів класу, такі як конструктори або деструктори, ігноруються.

Словники методів та станів trait'а позначатимемо як d_t та s_t відповідно. Тоді trait $t \in T$ визначається як $t = \langle s_t, d_t \rangle$, що є аналогічним до визначення класу. Композиція словників визначається аналогічно до існуючої моделі як для станів, так і для методів. Операцію аліасу можна застосовувати до словників станів. Проте операція перевантаження виконується лише для словників методів. Існуюча модель за замовчуванням передбачає, що при композиції trait'а в клас така операція має виконуватися для всіх методів. Це призводить до неявних перевантажень, що може призвести до помилок під час виконання. Для запобігання цьому пропонується окремо визначати набір методів trait'а, які можуть бути перевантаженими при композиції. Для цього введемо $o_t \in O$, де $o_t : N_m \to B_m^*$ і $o_t^{-1}(\mathbb{U}) = \varnothing$, а $o_t^{-1}(B_m)$ є скінченною. Тоді trait можна подати як $t = \langle s_t, d_t, o_t \rangle$, де $(d_t + o_t)^{-1}(\mathbb{C}) = \varnothing$. Визначимо операцію $\cup$ за виразом (4), що описує композицію trait і класу:

$$c \cup t = \langle s + s_t, (d \triangleright o_t) + d_t \rangle. \tag{4}$$

У низці мов програмування traits реалізовано шляхом додавання коду trait'а в код класу. Для цього визначено спеціальні ключові слова та процедури компілятора. Цього можна досягти за допомогою метапрограмування, використовуючи методи метапрограмування, що опираються на аналіз існуючого вихідного коду та генерацію нового коду за результатами аналізу. Це дозволить генерувати визначення класів, що міститимуть члени відповідних trait'ів. Пропонуємо використовувати модель роботи генератора коду (рис. 1).

Така модель визначає спеціальний генератор коду, що спрацьовує як на етапі компіляції, так і при змінах у вихідному коду. Це дозволить здійснювати генерацію під час розробки без необхідності компіляції.

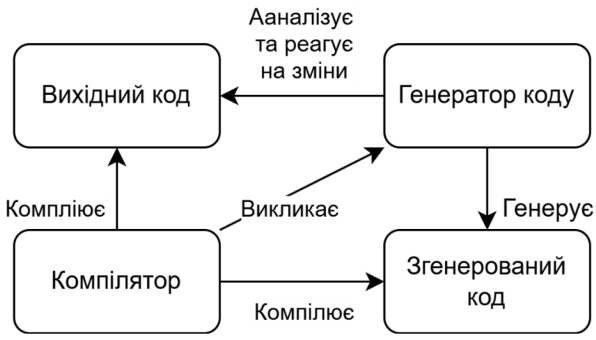

Рис. 1. Загальна модель роботи генератора коду

Найбільш доцільним інструментом, що дозволить забезпечити модель генерації (рис. 1), є інструмент Incremental Source Generators. Цей інструмент працює на рівні аналізатора коду, що дозволяє йому реагувати на зміни без необхідності компіляції. Разом з цим Incremental Source Generators дозволяють забезпечити хороші показники швидкодії генерації за рахунок використання інкрементного методу генерації.

Далі необхідно вирішити завдання визначення trait'а та класу, в який цей trait включається. Оскільки платформа .NET та мова програмування C# не підтримують traits, то відповідні ключові слова відсутні. Для вирішення цієї проблеми можна використовувати атрибути метаданих, адже доступ до них можливий як на етапі аналізу коду, так і під час його виконання. Таким чином, можна визначити атрибути метаданих для позначення типу даних як trait'а та класу, що включає в себе певний trait.

Далі варто визначити вимоги до типів даних trait'ів та класів, що їх включають. Виходячи з особливостей концепту traits та запропонованої моделі, основними вимогами до типу даних trait'а є:

– відсутність наслідування;
– можливість реалізовувати методи;
– можливість визначати метод без реалізації;
– відсутність можливості інстанціювання.

Отже, для визначення trait'а найбільш доцільним є використання абстрактного класу, який не бере участі в ієрархії наслідування. Щодо класу, в який включається trait, то ключовою вимогою є те, що такий клас має часткове об'явлення (partial). Це пов'язано із тим, що Incremental Source Generators не дозволяють вносити зміни в існуючий код, а передбачають лише генерацію нового коду.

Використання partial класів дозволить генерувати файл з частиною об'явлення класу, що містить члени відповідного trait'а (рис. 2).

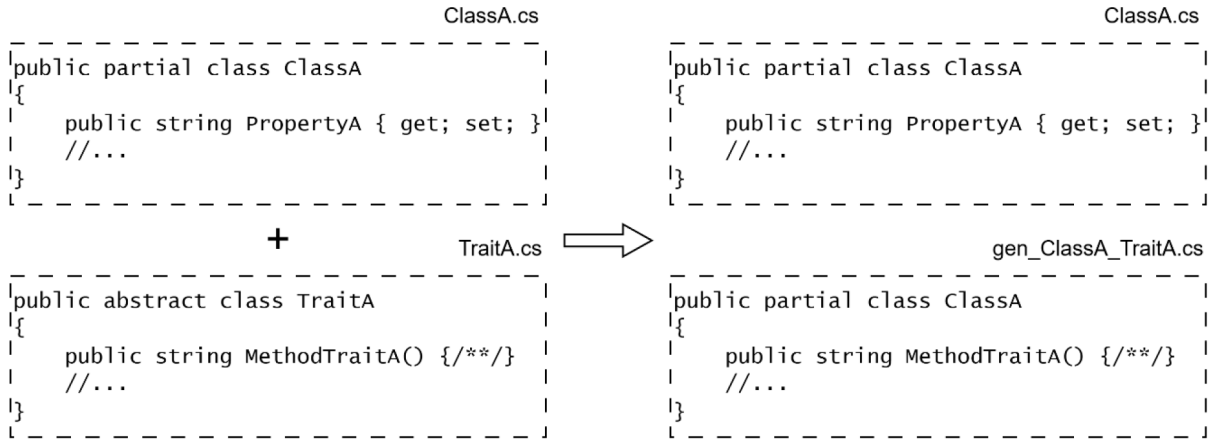

Рис. 2. Модель композиції trait'а і класу

Таким чином, композиція trait'а і класу відбуватиметься за рахунок генерації часткового об'явлення класу, що міститиме члени відповідного trait'а. Така модель дозволить повною мірою реалізувати особливості концепту traits.

Висновки

Запропоновано загальний метод реалізації traits в .NET з використанням методів метапрограмування. Метод використовує розширену модель trait's, що дозволяє визначати стани та контролювати перевантаження методів. Запропонований метод опирається на використання методів метапрограмування, що дозволяють генерацію

вихідного коду під час розробки та на етапі компіляції для генерації часткових об'явлень класів, що містять члени trait'ів, які вони включають. Для контролю поведінки генератора коду пропонується використовувати атрибути метаданих. Розроблений метод дозволить реалізувати концепт traits для платформи .NET та мови програмування C#, що сприятиме покращенню можливостей повторного використання програмних компонентів та модульності розроблюваного програмного забезпечення.

Список використаної літератури

1. Albalooshi F., Mahmood A. A Comparative Study on the Effect of Multiple Inheritance Mechanism in Java, C++, and Python on Complexity and Reusability of Code. *International Journal of Advanced Computer Science and Applications*. 2017. Т. 8, № 6. URL: https://doi.org/10.14569/ijacsa.2017.080614 (дата звернення: 20.08.2025).

2. Traits: Composable Units of Behaviour / N. Schärli та ін. *ECOOP 2003 – Object-Oriented Programming*. Berlin, Heidelberg, 2003. С. 248–274. URL: https://doi.org/10.1007/978-3-540-45070-2_12 (дата звернення: 20.08.2025).

3. Cassou D., Ducasse S., Wuyts R. Traits at work: The design of a new trait-based stream library. *Computer Languages, Systems & Structures*. 2009. Т. 35, № 1. С. 2–20. URL: https://doi.org/10.1016/j.cl.2008.05.004 (дата звернення: 20.08.2025).

4. A new modular implementation for stateful traits / P. Tesone та ін. *Science of Computer Programming*. 2020. Т. 195. С. 102470. URL: https://doi.org/10.1016/j.scico.2020.102470 (дата звернення: 20.08.2025).

5. Incremental Roslyn Source Generators In.NET6. URL: https://www.thinktecture.com/net/roslyn-source-generators-introduction/ (дата звернення: 20.08.2025).

6. Saadatmand M. Towards Automating Integration Testing of.NET Applications Using Roslyn. *2017 IEEE International Conference on Software Quality, Reliability and Security Companion (QRS-C)*, м. Prague, Czech Republic, 25–29 лип. 2017 р. 2017. URL: https://doi.org/10.1109/qrs-c.2017.99 (дата звернення: 20.08.2025).

7. Weber D., Christi A. Redusharptor: A Tool to Simplify Developer-Written C# Unit Tests. *International Journal of Software Engineering & Applications*. 2023. Т. 14, № 5. С. 29–40. URL: https://doi.org/10.5121/ijsea.2023.14503 (дата звернення: 20.08.2025).

8. Immutype. URL: https://github.com/DevTeam/Immutype (дата звернення: 20.08.2025).

9. AutoInterface. URL: https://github.com/beakona/AutoInterface (дата звернення: 20.08.2025).

10. Traits: A mechanism for fine-grained reuse / S. Ducasse та ін. *ACM Transactions on Programming Languages and Systems*. 2006. Т. 28, № 2. С. 331–388. URL: https://doi.org/10.1145/1119479.1119483 (дата звернення: 20.08.2025).

References

1. Albalooshi, F., & Mahmood, A. (2017). A Comparative Study on the Effect of Multiple Inheritance Mechanism in Java, C++, and Python on Complexity and Reusability of Code. *International Journal of Advanced Computer Science and Applications*, *8*(6). https://doi.org/10.14569/ijacsa.2017.080614

2. Schärli, N., Ducasse, S., Nierstrasz, O., & Black, A. P. (2003). Traits: Composable Units of Behaviour. У *ECOOP 2003 – Object-Oriented Programming* (с. 248–274). Springer Berlin Heidelberg. https://doi.org/10.1007/978-3-540-45070-2_12

3. Cassou, D., Ducasse, S., & Wuyts, R. (2009). Traits at work: The design of a new trait-based stream library. *Computer Languages, Systems & Structures*, *35*(1), 2–20. https://doi.org/10.1016/j.cl.2008.05.004

4. Tesone, P., Ducasse, S., Polito, G., Fabresse, L., & Bouraqadi, N. (2020). A new modular implementation for stateful traits. *Science of Computer Programming*, *195*, 102470. https://doi.org/10.1016/j.scico.2020.102470

5. Incremental Roslyn Source Generators In.NET 6. (2022). https://www.thinktecture.com/net/roslyn-source-generators-introduction/

6. Saadatmand, M. (2017). Towards Automating Integration Testing of.NET Applications Using Roslyn. У *2017 IEEE International Conference on Software Quality, Reliability and Security Companion (QRS-C)*. IEEE. https://doi.org/10.1109/qrs-c.2017.99

7. Weber, D., & Christi, A. (2023). Redusharptor: A Tool to Simplify Developer-Written C# Unit Tests. *International Journal of Software Engineering & Applications*, *14*(5), 29–40. https://doi.org/10.5121/ijsea.2023.14503

8. Immutype. (2025). https://github.com/DevTeam/Immutype

9. AutoInterface. (2025). https://github.com/beakona/AutoInterface

10. Ducasse, S., Nierstrasz, O., Schärli, N., Wuyts, R., & Black, A. P. (2006). Traits: A mechanism for fine-grained reuse. *ACM Transactions on Programming Languages and Systems*, *28*(2), 331–388. https://doi.org/10.1145/1119479.1119483

Дата першого надходження рукопису до видання: 26.09.2025
Дата прийнятого до друку рукопису після рецензування: 23.10.2025
Дата публікації: 28.11.2025