Відкриті інформаційні та комп'ютерні інтегровані технології, № 106, 2025 ISSN 2071-1077(print)
ISSN 2663-2411(online)

doi: 10.32620/oikit.2025.106.10

УДК 004.42

І. В. Шевченко,
П. О. Лучшев,
О. В. Лучшева

Дослідження підходів реалізації run-time поліморфізму у мові програмування C#

Національний аерокосмічний університет «Харківський авіаційний інститут»

У статті досліджено підходи реалізації динамічного (run-time) поліморфізму у мові програмування C#. Поліморфізм є одним із ключових принципів об'єктно-орієнтованого програмування, що забезпечує гнучкість, повторне використання та розширюваність програмного коду. Робота акцентує увагу на різних механізмах реалізації поліморфної поведінки, зокрема через віртуальні методи, абстрактні класи, інтерфейси, делегати та тип dynamic. Виконано порівняльний аналіз зазначених підходів, визначено їхні переваги, обмеження та практичні сфери застосування. Наведені приклади програмного коду демонструють роботу кожного механізму у реальних сценаріях, що дозволяє оцінити доцільність їх використання залежно від вимог конкретного програмного завдання. Особливу увагу приділено питанням забезпечення принципів SOLID та гнучкості архітектури програмного забезпечення. У результаті сформульовано рекомендації щодо вибору оптимального механізму реалізації динамічного поліморфізму в різних контекстах. Отримані висновки можуть бути корисними як для студентів і викладачів, так і для практикуючих розробників програмного забезпечення.

Ключові слова: C#; об'єктно-орієнтоване програмування; поліморфізм; run-time; віртуальні методи; абстрактні класи; абстрактні методи; інтерфейси; делегати; dynamic.

Вступ

У сучасній інженерії програмного забезпечення здатність створювати програмні системи, які є водночас надійними та гнучкими до змін, має першочергове значення. Об'єктно-орієнтоване програмування (ООП) [1-8] – потужна парадигма для забезпечення управління складністю програмного забезпечення, в основі якої лежить чотири базових принципи: інкапсуляція, наслідуванням, поліморфізм та абстракція. Найбільш багатогранним принципом з різними підходами для його реалізації є поліморфізм [1, 2, 4-9], назва якого в перекладі з грецької мови буквально означає «багато форм», що у програмуванні означає здатність об'єктів одного типу поводитися по-різному залежно від контексту. Ця здатність є фундаментальною для мови C# (статично типізованої, багатопарадигмальної), яка значною мірою використовує принципи ООП для створення масштабованих програм.

Важливо розрізняти поліморфізм під час компіляції (compile-time або статичний) та під час виконання (run-time або динамічний). Compile-time поліморфізм, відомий як статичне зв'язування, визначається під час компіляції і проявляється у перевантаженні методів або операторів. У цих випадках компілятор визначає, який метод викликати, виходячи з кількості та типів аргументів. На відміну від поліморфізму під час компіляції, поліморфізм під час виконання [4, 5, 7, 9], або динамічне зв'язування, відкладає це рішення до моменту роботи програми. Динамічна природа поліморфізму є ключовою для сценаріїв, коли точний тип об'єкта невідомий до часу виконання, що забезпечує більшу гнучкість і розширюваність.

Для реалізація run-time поліморфізму мова C# пропонує багатий набір конструкцій, кожна з яких має свої особливості та сфери застосування. Найпоширеніші підходи ґрунтуються на наслідуванні, використанні віртуальних методів [4-8] або абстрактних методів [4-8] для створення спеціалізованих поведінкових сценаріїв у межах ієрархії класів. Альтернативною й широко застосовуваною стратегією є інтерфейси [1, 2, 4-8], які визначають поведінковий контракт, що може бути реалізований будь-яким класом незалежно від його місця в ієрархії, завдяки чому забезпечується гнучкість у проєктуванні. Окрім цих базових механізмів, C# також підтримує поліморфізм через інші механізми, зокрема делегати [4, 6-8] для безпечного з погляду типів виклику методів та узагальнені типи (generics) для параметризованої гнучкості, а також гнучкого, але небезпечного поліморфізму через використання типу dynamic [7,8].

Реалізація поліморфізму під час виконання в C# нерозривно пов’язана з архітектурою рушія виконання платформи .NET – Common Language Runtime (CLR) [1, 4, 6-8]. CLR відповідає за управління виконанням .NET-застосунків та забезпечує механізми для динамічного виклику методів, також відомого як пізнє зв’язування. Коли програма компілюється, компілятор C# вбудовує у збірку метадані, які описують типи, їхні члени та взаємозв’язки, включно з інформацією про віртуальні або абстрактні методи та їх перевизначення.

Для ефективного проєктування програмного забезпечення необхідне глибоке розуміння переваг, продуктивності та архітектурних компромісів наведених реалізації run-time поліморфізму у мові C#.

1. Мета роботи і постановка завдань

Попри велику кількість робіт, присвячених об’єктно-орієнтованому програмуванню, питання практичної реалізації різних форм динамічного поліморфізму в мові C# часто розглядається фрагментарно. Більшість досліджень акцентують увагу на класичному перевизначенні методів, тоді як менш поширені механізми (делегати, динамічний тип dynamic) залишаються поза увагою.

Метою статті є систематизація та порівняльний аналіз основних механізмів реалізації динамічного (run-time) поліморфізму в C#, визначення їхніх переваг, недоліків та сфер ефективного застосування у розробленні програмного забезпечення.

Для досягнення цієї мети необхідно виконати такі завдання:

1. Проаналізувати особливості реалізації різних підходів реалізації run-time поліморфізму у мові C#: віртуальні і абстрактні методи, інтерфейси, делегати та динамічний тип dynamic.
2. Оцінити гнучкість і доцільність використання кожного з механізмів у контексті вирішення різних програмних завдань.
3. Сформулювати узагальнені висновки щодо оптимального використання динамічного поліморфізму в C#.

2. Реалізація поліморфізму за допомогою віртуальних методів

Дана реалізації є найбільш поширеним способом реалізації динамічного поліморфізму і полягає у поєднанні двох ключових слів virtual та override в ієрархії класів. Якщо метод у базовому класі оголошується як virtual, це означає, що він має реалізацію, яка може бути перевизначена більш специфічною реалізацією в

 ISSN 2071-1077(print)
ISSN 2663-2411(online)

будь-якому класі, який його наслідує. Похідні класи використовують ключове слово override, щоб надати цю спеціалізовану реалізацію.

Наведемо приклад ієрархії класів з використанням віртуального методу GetArea():

```
public class Shape
{
   // віртуальний метод
   public virtual double GetArea() => 0;    // базова реалізація (може бути умовною або пустою)
}

public class Circle : Shape
{
   public double Radius { get; set; }
   public Circle(double r) => Radius = r;

   // перевизначення віртуального методу
   public override double GetArea() => Math.PI * Radius * Radius;
}

public class Square : Shape
{
   public double Side { get; set; }
   public Square(double s) => Side = s;

   // перевизначення віртуального методу
   public override double GetArea() => Side * Side;
}
```

Реалізація поліморфізму за допомогою віртуальних методів забезпечує можливість працювати з об’єктами через посилання на базовий клас, не знаючи їх точного типу.

Продемонструємо використання потужної сили механізму поліморфізму з використанням побудованої ієрархії класів – для кожного об’єкта колекції shapes розрахуємо площу фігури за відповідною формулою:

```
class Program
{
   static void Main()
   {
      List<Shape> shapes = new List<Shape>     // список посилань на базовий клас
      {
         new Circle(5), new Square(4), new Circle(3), new Square(7)
      };

      foreach (var shape in shapes)
        Console.WriteLine(shape.GetArea());      // поліморфний виклик методу GetArea()
    }
}
```

Розберемо, що буде відбуватися при виклику метода shape.GetArea(). Змінна shape має статичний тип Shape (тобто компілятор бачить її як змінну базового класу), але реальний (динамічний) тип об’єкта у пам’яті – це тим Circle або Square. Метод GetArea() у базовому класі оголошений як virtual, а в дочірніх (Circle, Square) – як override. Це означає, що виклик методу буде пізньозв’язаний

(late binding), тобто вирішуватиметься не на етапі компіляції, а під час виконання.

CLR (Common Language Runtime) використовує механізм таблиць віртуальних методів (vtable) для реалізації динамічного поліморфізму:

- у кожного класу, що має virtual/override методи, є таблиця vtable вказівників на методи;
- коли створюються об’єкт new Circle(5), CLR заповнює vtable посиланнями на методи класу Circle;
- коли створюється об’єкт new Square(4), CLR заповнює vtable посиланням на методи класу Square;
- при виклику shape.GetArea() CLR дивиться на реальний тип об’єкта (Circle або Square), звертається до його таблиці методів і виконує правильний метод – Circle.GetArea() або Square.GetArea().

Варто зазначити, що в C# цей механізм реалізується не буквально через класичну vtable, а через внутрішні method table у CLR, які є конкретною реалізацією концепції віртуальної таблиці методів. Це дозволяє забезпечити як високу швидкість виклику, так і гнучкість у реалізації поліморфної поведінки.

Завдяки описаному механізму код у циклі foreach(…) не потребує умовних операторів для перевірки типу змінної shape, наприклад, if (shape is Circle)…, а виклик методу йде напряму через таблицю вказівників.

Віртуальний метод обов’язково має базову реалізацію, яка зазвичай підходить для більшості нащадків, але за потреби її можна змінити у специфічних випадках. Таким чином, використовуючи в базовому класі віртуальний метод гарантується наявність базової реалізації віртуального метода у похідних класах.

Однак у наведеному прикладі для віртуального методу GetArea() в базовому класі використовувалася умовна реалізація: фігура без конкретної форми буде мати площу, яка дорівнює 0. Саме зобов’язання мати базову реалізацію і вимагає зробити в даному випадку умовну реалізацію, яка в деяких випадках може бути доволі «штучною» лише для того, щоб задовольнити вимогу компілятора. У наступному розділі буде продемонстровано, як можна позбутися цієї умовної реалізації.

Зазначимо, що для розширення наведеного коду, наприклад, для додавання нової фігури Triangle треба додати новий клас-нащадок і в ньому перевизначити метод GetArea() з реалізацію відповідної формули розрахунку площі трикутника. Таким чином наявний код не буде потребувати зміни, а значить буде відповідати SOLID-принципу відкритості/закритості (Open/Closed Principle, OCP): «Класи повинні бути відкриті для розширення, але закриті для модифікації».

Однак потреба в додаванні нової функціональності, наприклад, метода GetPerimetr() буде потребувати зміни коду у базовому класі Shape() і перевизначення нового методу у класах-нащадках. У такому випадку OCP-принцип «ламається», бо доводиться змінювати базовий клас і всі похідні класи.

Ця проблема відома в ООП як «дилема OCP-принципу»: відкритість для розширення по одній осі (нові типи) обмежує відкритість по іншій осі (нові операції).

3. Реалізація поліморфізму за допомогою абстрактних методів

Абстрактний клас – це клас, екземпляр (об’єкт) якого не можна створити напряму (через оператор new), а можна лише цей клас унаслідувати. Головна причина заборони створювати екземпляри – це можлива наявність абстрактних

 ISSN 2071-1077(print)
ISSN 2663-2411(online)

методів.

Абстрактні методи – це такі методи, які не мають реалізації (тіла), а мають тільки сигнатуру (заголовок). Абстрактні методи позначаються ключовим словом abstract і вимагають обов’язкової реалізації в похідних класах з використанням ключового слова override, що гарантує, що кожен похідний клас буде мати власну реалізацію поведінки, визначеної цим методом. Таким чином абстрактні методи забезпечують жорстку контрактність: неможливо створити повноцінний (неабстрактний) клас-нащадок, не реалізувавши всі абстрактні методи базового класу.

Абстрактні методи є різновидом віртуальних методів і тому вони завжди мають бути перевизначені у похідних класах, але вони не можуть мати власної реалізації, і це дозволяє вирішити проблему з умовною (return 0;) або пустою реалізації (throw new NotImplementedException()). І якщо відомо, що без конкретного перевизначення метод не має сенсу, то його краще робити абстрактним:

```
public abstract class Shape
{
   // абстрактний метод
   public abstract double GetArea();
}

public class Circle : Shape
{
   public double Radius { get; set; }
   public Circle(double r) => Radius = r;

   // реалізація абстрактного методу
   public override double GetArea() => Math.PI * Radius * Radius;
}

public class Square : Shape
{
   public double Side { get; set; }
   public Square(double s) => Side = s;

   // реалізація абстрактного методу
   public override double GetArea() => Side * Side;
}
```

Для абстрактних методів, як і для віртуальних, CLR також будує таблицю диспетчеризації методів (method table), яка є реалізацією концепції vtable.

Код розрахунку для кожного об’єкта колекції shapes площі фігури за відповідною формулою повністю ідентичний попередньому прикладу:

```
class Program
{
   static void Main()
   {
      List<Shape> shapes = new List<Shape>     // список посилань на об’єкти-нащадки
      {
         new Circle(5), new Square(4), new Circle(3), new Square(7)
      };

      foreach (var shape in shapes)
        Console.WriteLine(shape.GetArea());    // поліморфний виклик методу GetArea()
```

ISSN 2071-1077(print)
ISSN 2663-2411(online)

```
    }
}
```

Вище вже зазначалося про те, що для абстрактних класів не можна створити їх екземпляри (об’єкти) через оператор new, однак на основі абстрактного класу Shape можна створити колекцію List<Shape>, в якій можна зберігати посилання тільки на об’єкти, які унаслідуються від Shape:

```
List<Shape> shapes = new List<Shape> // Список посилань на об’єкти-нащадки
{
     new Circle(5), new Square(4), new Circle(3), new Square(7)
};
```

Зазначимо, що якщо базовий клас Shape не є абстрактним, то на його основі можна створити колекцію List<Shape>, в яку дозволяється додавати посилання як на базовий клас, так і на об’єкти його нащадків:

```
List<Shape> shapes = new List<Shape> // Список посилань на базовий клас
{
    new Shape(), new Circle(5), new Square(4), new Circle(3), new Square(7)
};
```

Недолік останнього коду полягає в наступному: якщо базовий клас Shape буде мати віртуальний метод зі «штучною» (умовною) реалізацією, то виклик цього методу для списку об’єктів може вести себе непередбачувано або некоректно в певному контексті на об’єктах базового класу.

Зазначимо, що абстрактні класи не дозволяють вирішити дилему OCP-принципу: нові типи додати в ієрархію можна без змін коду, але додати нову функціональність (метод) без зміни існуючого коду буде неможливо. Спробуємо розв’язати цю проблему за допомогою використання інтерфейсів.

4. Реалізація поліморфізму за допомогою інтерфейсів

Інтерфейс у C# – це контракт, що визначає набір членів (методів, властивостей, індексаторів, подій), які клас (або структура) зобов’язані реалізувати. Клас, який реалізує інтерфейс, бере на себе відповідальність за забезпечення цієї поведінки.

У мові C# класу (структурі) на відміну від наслідування дозволяється реалізовувати декілька інтерфейсів одночасно, що забезпечує гнучкий механізм поліморфізму без наслідування.

Наведемо реалізацію прикладу з фігурами через механізм інтерфейсів. Замість базового класу визначимо інтерфейс IShape, який міститиме опис методу GetArea(). Кожен клас, що реалізує інтерфейс IShape, зобов’язаний надати власну реалізацію цього методу. Таким чином гарантується наявність реалізації методу GetArea() у всіх необхідних класах:

```
public interface IShape
{
   // метод інтерфейсу
   double GetArea();
}
```

Класи, які будуть реалізовувати інтерфейс IShape мають обов’язково

 ISSN 2071-1077(print)
ISSN 2663-2411(online)

реалізувати всі методи цього інтерфейсу, тобто надати тіло методу GetArea(), або такі класи мають бути визначені як абстрактні:

```
public class Circle : IShape
{
   public double Radius { get; set; }
   public Circle(double r) => Radius = r;

   // реалізація інтерфейсного методу
   public double GetArea() => Math.PI * Radius * Radius;
}

public class Square : IShape
{
   public double Side { get; set; }
   public Square(double s) => Side = s;

   // реалізація інтерфейсного методу
   public double GetArea() => Side * Side;
}
```

Зміни в основному коді відносно попередніх прикладів полягають у створенні колекції List<IShape>, яка представляє собою список посилань на об’єкти класів, які реалізують інтерфейс IShape, а значить ці класи взяли на себе зобов’язання по реалізації методу GetArea():

```
class Program
{
   static void Main()
   {
      // список посилань на об’єкти класів, які реалізують інтерфейс IShape
      List<IShape> shapes = new List<IShape>
      {
         new Circle(5), new Square(4), new Circle(3), new Square(6)
      };

      foreach (var shape in shapes)
         Console.WriteLine(shape.GetArea());  // поліморфний виклик методу GetArea()
   }
}
```

Динамічний поліморфізм через інтерфейси працює наступним чином: кожен клас, що реалізує інтерфейс, має таблицю методів інтерфейсу (interface method table), яку CLR використовує для диспетчеризації викликів. За принципом роботи вона подібна до vtable для віртуальних методів: коли викликається shape.GetArea(), CLR дивиться на фактичний тип об’єкта, на який посилається shape, знаходить у таблиці реалізацію GetArea() цього класу – Circle.GetArea() або Square.GetArea() – і виконує виклик.

Розглянемо схожість та відмінність у використанні інтерфейсів та абстрактних класів у C#:

```
public abstract class Shape
{
   public abstract double GetArea();
}

public interface IShape
```

 ISSN 2071-1077(print)
ISSN 2663-2411(online)

```
{
   double GetArea();
}
```

Схожість полягає в наступному: 1) абстрактний клас може містити абстрактні методи, які зобов'язують похідні класи реалізувати їх; інтерфейс також визначає методи та властивості, які обов'язково реалізуються в класі, що його наслідує; 2) через базовий абстрактний клас або інтерфейс можна працювати з об'єктами різних конкретних класів, викликаючи однакові методи.

Відмінність полягає в тому, що абстрактні методи не можуть мати реалізації за замовчуванням, тоді як у інтерфейсі (починаючи з версії C# 8.0) методи/властивості можуть її мати.

Однак принципова відмінність полягає в тому, що клас може унаслідувати лише один базовий клас, але реалізовувати необмежену кількість інтерфейсів одночасно. Саме ця особливість інтерфейсів дає можливість замість одного «товстого» інтерфейсу IShape визначити окремі інтерфейси, наприклад, IAreaCalculable, IPerimeterCalculable, IDrawable, і реалізовувати їх у тих класах, для яких це дійсно актуально. Така реалізація буде відповідати SOLID-принципу ISP (Interface Segregation Principle), що робить ієрархію гнучкішою та дозволяє додавати нові можливості без модифікації існуючих класів.

Тут варто зазначити, що у разі необхідності додавання нової функціональності (методу), яку не покриває існуючий інтерфейс, її можна реалізувати через створення нового інтерфейсу або зміни базового контракту. При додаванні нової функціональності в існуючий клас, треба в перелік інтерфейсів класу додати новий інтерфейс, наприклад, IPerimeterCalculablea, а значить необхідні будуть зміни в початковому коді класу через необхідність додавання реалізацій методів нового інтерфейсу. Тобто «дилема OCP-принципу» залишається невирішеною, повне дотримання принципу OCP по всіх осях не завжди можливо.

Також зазначимо, що інтерфейси забезпечують гнучкість і слабке зв'язування (loose coupling) класів, що відповідає принципу інверсії залежностей DIP (Dependency Inversion). Для демонстрації відмінності між жорстким і слабким зв'язуванням класів розглянемо приклад жорсткого зв'язування:

```
public class AreaPrinter
{
   public void PrintArea(Circle circle)
   {
      Console.WriteLine(circle.GetArea());
   }
}
```

У цьому прикладі клас AreaPrinter жорстко пов'язаний із конкретною реалізацією фігур – класом Circle. Метод PrintArea() може працювати тільки з об'єктами класу Circle, оскільки його параметр жорстко типізований як Circle. Якщо виникне потреба друкувати площу інших фігур, наприклад Square, доведеться змінювати код класу AreaPrinter, додаючи нові перевантаження методу PrintArea() для кожного нового типу. Це порушує принцип Open/Closed (OCP), за яким класи мають бути відкриті для розширення, але закриті для модифікації.

Цю проблему можна вирішити, забезпечивши слабке зв'язування класів:

 ISSN 2071-1077(print)
ISSN 2663-2411(online)

якщо метод PrintArea() буде приймати параметр типу абстрактного класу Shape або інтерфейсу IShape. У випадку використання абстрактного класу можна буде передавати лише об’єкти класів-нащадків цього класу, а використання інтерфейсу дозволить методу приймати будь-які об’єкти, які реалізують IShape, навіть якщо вони не належать до ієрархії класів фігур.

Змінимо у прикладі тип параметра методу PrintArea() на інтерфейс IShape:

```
public class AreaPrinter
{
   public void PrintArea(IShape shape)
   {
      Console.WriteLine(shape.GetArea());
   }
}
```

Тепер метод PrintArea() не залежить від конкретного типу (наприклад, Circle чи Square), що дозволяє легко розширювати функціональність програми обробкою нових фігур, які реалізують інтерфейс IShape, або навіть об’єктів класів, які не мають відношення до фігур, але реалізують інтерфейс IShape. Таким чином використання інтерфейсу забезпечує більш гнучкий варіант слабкого зв’язування в порівнянні із застосуванням абстрактних класів.

Даний приклад реалізує SOLID-принцип DIP (Dependency Inversion Principle): «Високорівневі модулі не повинні залежати від низькорівневих. Обидва повинні залежати від абстракцій», «Абстракції не повинні залежати від деталей. Деталі повинні залежати від абстракцій». Таким чином, код не повинен жорстко залежати від конкретних класів, гнучкість забезпечується використанням абстракцій (інтерфейсів або абстрактних класів). У наведеній реалізації високорівневий клас AreaPrinter залежить від абстракції IShape і низькорівневі класи Circle та Square також залежать від тієї ж абстракції. Таким чином забезпечується слабке зв’язування та можливість легко додавати нові фігури без модифікації високорівневого модуля.

Про слабке зв’язування варто зазначити також, що воно полегшує unit-тестування завдяки можливості використання mock-об’єктів (об’єктів-заглушок), які імітують поведінку реальних об’єктів і спрощують тестування програми:

```
public class MockShape : IShape
{
   private double _area;
   public MockShape(double area) => _area = area;
   public double GetArea() => _area;      // повертає заздалегідь задане значення
}

// реалізація тесту
var mock = new MockShape(45);          // створення об’єкта-заглушки зі значенням площі 45
var printer = new AreaPrinter();
printer.PrintArea(mock);               // використання об’єкта-заглушки
```

Тут клас AreaPrinter працює з абстракцією IShape, тож для тестування можна підставляти будь-які об’єкти, що реалізують інтерфейс IShape, без залежності від конкретних реалізацій (Circle, Square).

Таким чином, інтерфейси є дуже гнучким механізмом, який забезпечує множинне контрактне наслідування, тоді як абстрактний клас забезпечує лише

 ISSN 2071-1077(print)
ISSN 2663-2411(online)

один шлях ієрархії.

Для вибору між абстрактним класом та інтерфейсом слід керуватися наступним принципом: якщо існує чітка ієрархія класів, то обирають абстрактний клас; якщо ієрархія не визначена чітко та необхідно реалізувати декілька контрактів одночасно, то обирають інтерфейс.

При розгляді інтерфейсів як засобу реалізації динамічного поліморфізму слід продемонструвати поєднання динамічного та параметричного поліморфізмів.

У C# використовуючи Generics можна писати універсальний код, який працює з будь-яким типом, підставленим як параметр <T>, без втрати продуктивності й без необхідності використовувати object та ручне перетворення типів (casting).

Generics (узагальнені типи) – це механізм, який дозволяє створювати типобезпечний і повторно використовуваний код (методи, класи, структури, інтерфейси), реалізуючи параметричний поліморфізм.

Розглянемо приклад класу ShapeProcessor з параметром <T>, де тип T має обмеження – обов’язкова реалізація інтерфейсу IShape:

```
public class ShapeProcessor<T> where T : IShape
{
   public double TotalArea(List<T> shapes)
   {
      double sum = 0;
      foreach (var s in shapes)
         sum += s.GetArea();     // поліморфний виклик методу GetArea()
      return sum;
   }
}
```

Наведений клас ShapeProcessor містить метод TotalArea(), який приймає список об’єктів будь-якого типу T з обмеженням where T : IShape. Оскільки будь-який тип T має реалізовувати інтерфейс IShape, то кожен об’єкт типу T гарантовано матиме реалізацію методу GetArea().

Таким чином наведений приклад демонструє, як Generics у поєднанні з інтерфейсами дозволяють будувати гнучкі, типобезпечні та поліморфні рішення.

5. Реалізація поліморфізму за допомогою делегатів

Окрім класичних підходів реалізації поліморфізму, у мові C# для реалізації поліморфної поведінки можна використовувати делегати.

Делегат – це спеціальний тип, змінна якого зберігає посилання на метод або список методів певної сигнатури; іншими словами, делегат є «типобезпечним вказівником на метод(и)».

За допомогою делегатів можна:

– передавати методи як аргументи іншим методам, що дозволяє робити програмний код більш гнучким і розширюваним;

– змінювати логіку виконання без необхідності перевантаження методів чи жорсткого «зашивання» поведінки;

– повертати методи з методів, оскільки делегат інкапсулює посилання на метод.

Розглянемо приклад реалізації поліморфної поведінки за допомогою делегату. За основу беремо реалізацію класів Circle і Square без спадкування:

 ISSN 2071-1077(print)
ISSN 2663-2411(online)

```
public class Circle
{
   public double Radius { get; set; }
   public Circle(double r) => Radius = r;
   public double GetArea() => Math.PI * Radius * Radius;
}
public class Square
{
   public double Side { get; set; }
   public Square(double s) => Side = s;
   public double GetArea() => Side * Side;
}
```

Наведемо визначення делегата AreaDelegate, а також змінимо основний код програми:

```
public delegate double AreaDelegate();

class Program
{
   static void Main()
   {
      List<AreaDelegate> areas = new List<AreaDelegate> // список делегатів
      {
         new Circle(5).GetArea,
         new Square(4).GetArea,
         new Circle(3).GetArea,
         new Square(6).GetArea
      };

      foreach (var area in areas)
        Console.WriteLine(area());   // поліморфний виклик через делегат
   }
}
```

Делегат AreaDelegate визначає єдиний контракт – метод без параметрів, який повертає double. Методи GetArea() у класах Circle і Square відповідають цьому контракту, тому їх можна «прикріпити» до делегата AreaDelegate. List<AreaDelegate> areas – це список делегатів, тобто список «вказівників на методи», де кожен метод має сигнатуру double MethodName(). Список можна заповнити методами GetArea() різних об’єктів (Circle, Square) і викликати їх через однаковий синтаксис area(). Виклик area() у циклі foreach(…) демонструє поліморфну поведінку: однакова форма виклику виконує різну логіку залежно від того, який метод інкапсулює делегат.

Зазначимо, що делегат може зберігати посилання на декілька методів одночасно та викликати їх послідовно при одному зверненні. Це також можна розглядати як прояв поліморфної поведінки.

Обгорткою над делегатами є події (events), які забезпечують безпечніший і більш контрольований механізм виклику методів. Якщо делегат може бути викликаний напряму, то подія інкапсулює цей процес, дозволяючи ззовні лише підписуватися (subscribe) або відписуватися (unsubscribe) від викликів, але не викликати їх безпосередньо. Такий підхід робить модель подій у C# більш захищеною та передбачуваною.

 ISSN 2071-1077(print)
ISSN 2663-2411(online)

6. Реалізація поліморфізму за допомогою типу dynamic

Мова C# підтримує ключове слово dynamic, яке визначає спеціальний тип даних, що відключає перевірку типів на етапі компіляції. Для звичайних типів (наприклад, int, string, object) коректність викликів методів та властивостей перевіряється під час компіляції, тоді як для dynamic – лише під час виконання програми (run-time).

Змінній типу dynamic можна присвоювати будь-який об’єкт, і компілятор не буде перевіряти, чи існують методи або властивості, до яких звертається код – це робиться вже під час виконання. Таким чином, тип dynamic забезпечує механізм пізнього зв’язування (late binding).

При роботі зі змінними типу dynamic середовище CLR залучає компонент DLR (Dynamic Language Runtime), який виконує динамічне визначення методів і властивостей об’єкта під час виконання програми, забезпечуючи механізм пізнього зв’язування.

Наведемо приклад того, як тип dynamic дозволяє реалізувати поліморфну поведінку, викликаючи методи, не знаючи точного типу об’єкта на момент компіляції:

```
public class Circle
{
   public double Radius { get; set; }
   public Circle(double r) => Radius = r;
   public double GetArea() => Math.PI * Radius * Radius;
}

public class Square
{
   public double Side { get; set; }
   public Square(double s) => Side = s;
   public double GetArea() => Side * Side;
}

class Program
{
   static void Main()
   {
      List<dynamic> shapes = new List<dynamic>  // Список динамічних об'єктів
      {
         new Circle(5), new Square(4), new Circle(3), new Square(7)
      };

      foreach (var shape in shapes)
        Console.WriteLine(shape.GetArea());  // поліморфний виклик методу GetArea()
   }
}
```

У наведеному прикладі використовується List<dynamic> shapes, який є списком динамічних об’єктів, тобто списком посилань на об’єкти, тип яких компілятор не перевіряє на момент компіляції. Виклики методів на елементах цього списку виконуються динамічно під час виконання програми. Виклик метода shape.GetArea() у циклі foreach(…) демонструє динамічний поліморфізм без використання наслідування або інтерфейсів. Об’єкти списку List<dynamic> shapes реалізують метод GetArea(), але не належать до спільного базового класу. Ключове слово dynamic дозволяє викликати методи незалежно від статичного

ISSN 2071-1077(print)
ISSN 2663-2411(online)

типу об’єкта.

Таким чином, тип dynamic дозволяє реалізовувати поліморфну поведінку без створення ієрархії класів або спільних інтерфейсів. Він спрощує інтеграцію з іншими платформами, наприклад, з COM-об’єктами, бібліотеками на основі Reflection або JSON, а також полегшує написання коду у ситуаціях, коли типи об’єктів невідомі заздалегідь.

Однак відсутність перевірки типів на етапі компіляції може призводити до помилок, які будуть виявлені лише під час виконання. У списку List<dynamic> shapes дозволяється зберігати посилання і на об’єкти, які не реалізують метод GetArea(), і це стане очевидним тільки під час виконання програми через викидання відповідного Exception і переривання виконання програми. Крім того, виклики через dynamic виконуються повільніше, оскільки включають механізми пізнього зв’язування, а також ускладнюють розуміння та підтримку коду.

Зазначимо, що поліморфізм через dynamic – нетиповий, але потужний інструмент, який дозволяє реалізовувати гнучку поведінку програм без суворого дотримання класичних ООП-принципів. Він відкриває можливості для динамічного програмування у C#, але вимагає обережності, оскільки втрачається безпека типів і частина переваг статичної типізації. Тим dynamic рекомендується використовувати лише там, де альтернативи (віртуальні і абстрактні методи, інтерфейси, делегати) надто обтяжливі або непридатні.

7. Рекомендації щодо застосування реалізацій run-time поліморфізму

Були розглянуті особливості реалізації у мові програмування C# поліморфної поведінки через віртуальні методи, абстрактні класи, інтерфейси, делегати і тип dynamic. Кожен із цих підходів має сильні та слабі сторони, які наведені у таблиці 1.

Таблиця 1

Сильні та слабкі сторони різних реалізацій run-time поліморфізму

Підхід	Сильні сторони	Слабкі сторони
Віртуальні методи	– наявність базової (default) реалізації і перевизначення за потреби; – розширюваність без зміни базового коду; – простота у використанні.	– залежність від ієрархії класів; – можливі небажані перевизначення; – у складній ієрархії з багатьма перевизначеннями складно відстежувати логіку коду; – відсутність контролю над обов’язковими перевизначеннями.
Абстрактні класи	– можливість часткової реалізації (наявність як абстрактних методів, так і звичайних або віртуальних методів з готовою логікою); – повний контроль над обов’язковими перевизначеннями абстрактних методів; – добре реалізують зв’язок «is-a».	– жорстка ієрархія; – обмежена гнучкість у наслідуванні (тільки один базовий клас); – складність модифікації базового класу через зміни у всіх нащадках; – дублювання коду при однакових перевизначеннях; – менша гнучкість у слабкому зв’язуванні.

 ISSN 2071-1077(print)
ISSN 2663-2411(online)

Продовження таблиці 1

Підхід	Сильні сторони	Слабкі сторони
Інтерфейси	– незалежність від ієрархій; – множинна реалізація; – чітке розділення контрактів; – слабке зв'язування; – ідеально підходять для Dependency Injection; підвищують тестованість коду.	– відсутність default-реалізації (до C# 8.0); – складність модифікації через потребу у змінах у всіх класах, які реалізують даний інтерфейс; велика кількість інтерфейсів робить клас складним для розуміння і підтримки.
Делегати	– можливість інкапсулювати метод; – дуже гнучкі та потужні для динамічних викликів і подій; – підтримка multicast (один делегат може містити декілька методів, які будуть викликатися підряд); – слабке зв'язування (класи взаємодіють, не знаючи деталей один одного); – підтримка анонімних методів і лямбда-виразів; – компактність.	– складно відстежувати потік виконання; – обмеженість у сигнатурі; – небезпека null-reference; – складність відлагодження.
Тип dynamic	– максимальна гнучкість при роботі з типами, визначеними під час виконання; – не потрібно приведення типів; – зручно для роботи з JSON, COM, зовнішніми бібліотеками та динамічними API.	– втрата безпеки типів; – помилки лише під час виконання; – зниження продуктивності; – складність відлагодження.

Віртуальні методи рекомендується використовувати, коли: 1) потрібна базова реалізація «за замовчуванням», яку більшість нащадків можуть унаслідувати без змін, а в окремих випадках – перевизначати; 2) існує багато спільного функціоналу для різних похідних класів, і потрібно уникати дублювання коду, залишаючи можливість гнучкого розширення; 3) поведінка об'єктів за замовчуванням є коректною і логічною, навіть якщо її не буде змінено в окремих підкласах; 4) потрібно зберегти зворотну сумісність: базовий клас може містити готову реалізацію, а нові нащадки зможуть її перевизначати без зміни існуючого коду.

Абстрактні методи рекомендується використовувати, коли:1) неможливо визначити адекватну базову реалізацію методу (наприклад, метод GetArea() для абстрактного класу Shape); 2) необхідно гарантувати, що всі нащадки обов'язково реалізують деякий метод, і при цьому треба уникнути «штучних» або некоректних реалізацій «за замовчуванням»; 3) базовий клас виконує роль чистої концептуальної моделі (каркасу), а не містить готову поведінку; 4) потрібно чітко розмежувати класи, які можна створювати напряму (конкретні), та ті, які слугують лише як основа для ієрархії (абстрактні).

ISSN 2071-1077(print)
ISSN 2663-2411(online)

Інтерфейси рекомендується використовувати, коли: 1) класи можуть мати різні ієрархії походження, але потребують спільної поведінки; 2) треба уникнути наслідування зайвої поведінки; 3) необхідно реалізувати множинну спадковість поведінки, що неможливо зробити через класи; 4) важлива гнучкість і слабке зв'язування між компонентами – залежність будується від контрактів, а не від конкретних реалізацій.

Делегати рекомендується використовувати, коли: 1) потрібно передавати метод як аргумент іншому методу (наприклад, Func<>, Action<>, Predicate<>); 2) необхідно реалізувати зворотний виклик (callback) у відповідь на певну дію; 3) важлива гнучка заміна алгоритму під час виконання без створення додаткових класів; 4) потрібно побудувати подійну модель (event-driven model), в якій виконання програми визначається не послідовним кодом зверху вниз, а подіями (натискання кнопки, отримання повідомлення, завершення завантаження файлу тощо); код «реагує» на події через обробники подій (event handlers), які найчастіше реалізуються через делегати.

Тип dynamic рекомендується використовувати, коли: 1) тип об'єкта евідомий на етапі компіляції і визначається лише під час виконання (наприклад, при роботі з JSON, COM-об'єктами, скриптовими мовами); 2) потрібно інтегруватися з зовнішніми або гнучкими API, які можуть змінюватися структуру даних динамічно; 3) гнучкість і швидкість розробки важливіша за максимальну продуктивність і безпеку типів.

8. Висновок

У статті виконано систематизацію та порівняльний аналіз основних підходів реалізації динамічного (run-time) поліморфізму у мові програмування C#. Розглянуто механізми на основі віртуальних та абстрактних методів, інтерфейсів, делегатів і типу dynamic. Для кожного з підходів наведено приклади реалізації, окреслено їхні переваги, обмеження та типові сфери застосування.

Отримані результати дають підстави стверджувати, що вибір конкретного механізму реалізації поліморфізму має ґрунтуватися на вимогах до архітектури програмного забезпечення: для класичних ієрархій доцільно застосовувати віртуальні та абстрактні методи; для забезпечення слабкого зв'язування і масштабованості – інтерфейси; для побудови подій та callback-логіки – делегати; для роботи з невідомими наперед структурами даних – тип dynamic.

Таким чином, комбіноване використання різних механізмів динамічного поліморфізму дозволяє створювати гнучкі, розширювані та підтримувані програмні системи, що відповідає сучасним вимогам до якості програмного забезпечення.

Список літератури

1. Sarcar V. Simple and Efficient Programming with C#: Skills to Build Applications with Visual Studio and .NET. – 2-е вид. – Berkeley, CA, USA : Apress, 2022. – 313 с. – DOI: https://doi.org/10.1007/978-1-4842-8737-8.

2. Panjuta D. Learning C# Through Small Projects / D. Panjuta, J. Jabbarzadeh. – 1st ed. – Cham, Switzerland : Springer, 2024. – 391 с. – DOI: https://doi.org/10.1007/978-3-031-51914-7.

3. Alls J. Clean Code with C#: Refactor your legacy C# code base and improve application performance using best practices. – Birmingham, UK : Packt Publishing

Ltd, 2023. – 492 с.

4. Bhalla S. S. Programming in C#: Exam 70-483 (MCSD) Guide / S. S. Bhalla, S. M. Gorthi. – 1-е вид. – Birmingham, UK: Packt Publishing, 2019. – 444 с.

5. Taher R. Hands-On Object-Oriented Programming with C#: Build maintainable software with reusable code using C#. – Бірмінгем, UK : Packt Publishing, 2019. – 288 с.

6. Bancila M., Rialdi R., Sharma A., Esposito D. Learn C# Programming: A guide to building a solid foundation in C# language for writing efficient programs. – Бірмінгем, UK : Packt Publishing, 2020. – 636 с.

7. Troelsen A., Japikse P. Pro C# 10 with .NET 6. – Берклі, CA, USA : Apress, 2022. – 1640 с. – DOI: https://doi.org/10.1007/978-1-4842-7869-7.

8. Price M. J. C# 13 and .NET 9 – Modern Cross-Platform Development Fundamentals: Start building websites and services with ASP.NET Core 9, Blazor, and EF Core 9. – 9-те вид. – Бірмінгем, UK : Packt Publishing, 2024. – 828 с.

9. Donchev I.Dynamic Polymorphism without Inheritance: Implications for Education / I. Donchev, E. Todorova // International Journal of Advanced Computer Science and Applications (IJACSA). – 2022. – Т. 13, № 10. – С. 643–649. – DOI: http://dx.doi.org/10.14569/IJACSA.2022.0131075.

References

1. Sarcar V. Simple and Efficient Programming with C#: Skills to Build Applications with Visual Studio and .NET. – 2nd ed. – Berkeley, CA, USA : Apress, 2022. – 313 p. – DOI: https://doi.org/10.1007/978-1-4842-8737-8.

2. Panjuta D. Learning C# Through Small Projects / D. Panjuta, J. Jabbarzadeh. – 1st ed. – Cham, Switzerland : Springer, 2024. – 391 p. – DOI: https://doi.org/10.1007/978-3-031-51914-7.

3. Alls J. Clean Code with C#: Refactor Your Legacy C# Code Base and Improve Application Performance Using Best Practices. – Birmingham, UK : Packt Publishing Ltd, 2023. – 492 p.

4. Bhalla S. S. Programming in C#: Exam 70-483 (MCSD) Guide / S. S. Bhalla, S. M. Gorthi. – 1st ed. – Birmingham, UK : Packt Publishing, 2019. – 444 p.

5. Taher R. Hands-On Object-Oriented Programming with C#: Build maintainable software with reusable code using C#. – Birmingham, UK : Packt Publishing, 2019. – 288 p.

6. Bancila M., Rialdi R., Sharma A., Esposito D. Learn C# Programming: A guide to building a solid foundation in C# language for writing efficient programs. – Birmingham, UK : Packt Publishing, 2020. – 636 p.

7. Troelsen A., Japikse P. Pro C# 10 with .NET 6. – Berkeley, CA, USA : Apress, 2022. – 1640 p. – DOI: https://doi.org/10.1007/978-1-4842-7869-7.

8. Price M. J. C# 13 and .NET 9 – Modern Cross-Platform Development Fundamentals: Start building websites and services with ASP.NET Core 9, Blazor, and EF Core 9. – 9th ed. – Birmingham, UK : Packt Publishing, 2024. – 828 p.

9. Donchev I. Dynamic Polymorphism without Inheritance: Implications for Education / I. Donchev, E. Todorova // International Journal of Advanced Computer Science and Applications (IJACSA). – 2022. – Vol. 13, No. 10. – P. 643–649. – DOI: http://dx.doi.org/10.14569/IJACSA.2022.0131075.

Надійшла до редакції 07.10.2025, розглянута на редколегії 10.11.2025

ISSN 2071-1077(print)
ISSN 2663-2411(online)

Research on Approaches to Implementing Run-Time Polymorphism in the C# Programming Language

The article examines approaches to implementing dynamic (run-time) polymorphism in the C# programming language. Polymorphism is one of the key principles of object-oriented programming, providing flexibility, reusability, and extensibility of program code. The study focuses on various mechanisms for implementing polymorphic behavior, in particular through virtual methods, abstract classes, interfaces, delegates, and the dynamic type. A comparative analysis of these approaches has been carried out, identifying their advantages, limitations, and practical areas of application. The provided code examples demonstrate the operation of each mechanism in real-world scenarios, allowing for the evaluation of their appropriateness depending on the requirements of a specific programming task. Special attention is given to ensuring compliance with the SOLID principles and the flexibility of software architecture. As a result, recommendations are formulated regarding the choice of the optimal mechanism for implementing dynamic polymorphism in different contexts. The findings may be useful for students and educators, as well as for practicing software developers.

Key words: C#; object-oriented programming; polymorphism; run-time; virtual methods; abstract classes; abstract methods; interfaces; delegates; dynamic.

Відомості про авторів:

Шевченко Ілона Володимірівна – к.т.н., доцент, кафедра інженерії програмного забезпечення, Національний аерокосмічний університет «Харківський авіаційний інститут». Ел. пошта: ilona.shevchenko@gmail.com, ORCID: https://orcid.org/0000-0003-0100-0726.

Лучшев Павло Олександрович – к.т.н., доцент, кафедра інженерії програмного забезпечення, Національний аерокосмічний університет «Харківський авіаційний інститут». Ел. пошта: p.luchshev@khai.edu, ORCID: https://orcid.org/0000-0002-3522-7622.

Лучшева Оксана Вадимівна – старший викладач, кафедра інженерії програмного забезпечення, Національний аерокосмічний університет «Харківський авіаційний інститут». Ел. пошта: o.luchsheva@khai.edu, ORCID: https://orcid.org/0000-0003-3855-2815.

About the Authors:

Ilona Shevchenko – Ph.D, associate professor of the software engineering department, National aerospace university «Kharkiv Aviation Institute». Email: ilona.shevchenko@gmail.com, ORCID: https://orcid.org/0000-0003-0100-0726.

Pavlo Luchshev – Ph.D, associate professor of the software engineering department, National aerospace university «Kharkiv Aviation Institute». Email: p.luchshev@khai.edu, ORCID: https://orcid.org/0000-0002-3522-7622.

Oksana Luchsheva – senior lecturer of the software engineering department, National aerospace university «Kharkiv Aviation Institute». Email: o.luchsheva@khai.edu, ORCID: https://orcid.org/0000-0003-3855-2815.