УДК 004.432.2, 004.422.83 *http://dx.doi.org/10.30970/vam.2026.36.00000*

ЗАСОБИ МОВ ПРОГРАМУВАННЯ ВИСОКОГО РІВНЯ ДЛЯ ГЛИБОКОГО КОПІЮВАННЯ ОБ'ЄКТІВ

С.А. Ярошко, С.М. Ярошко

Львівський національний університет імені Івана Франка,
вул. Університетська 1, Львів, 79000, Україна,
e-mail: serhiy.yaroshko@lnu.edu.ua

Виконано порівняння можливостей мов програмування C++, C#, Python і Pharo щодо копіювання об'єктів. Якщо до складу об'єкта входить хоча б одне поліморфне посилання (або вказівник), задача глибокого копіювання перестає бути тривіальною. Як модельний приклад використано послідовний контейнер, який містить різнотипні об'єкти, що моделюють геометричні фігури. Показано, як кожною з мов оголосити відповідні класи, наповнити колекцію та побудувати її копію. Виконані перевірки засвідчили, що отримана копія не залежить від оригіналу.

Мовою C++ глибоке копіювання виконано "ручним" способом: на базі *std::vector* оголошено шаблон поліморфного вектора, який коректно очищає пам'ять від динамічних об'єктів, правильно виконує копіювання вмісту контейнера з використанням спеціальних методів, оголошених у класах фігур. Копіювання мовою C# виконано за допомогою загальних методів LINQ, а мовами Python і Pharo – стандартними вбудованими засобами. Показано, що Pharo надає програмістові найширше коло можливостей щодо копіювання об'єктів.

Ключові слова: глибоке копіювання, поверхневе копіювання, поліморфна колекція, конструктор копіювання, інтерфейс *ICloneable*, модуль *copy*, метод *postCopy*.

1. Вступ

Екземпляри класів, об'єкти, у сучасних програмах зазвичай розташовуються в динамічній пам'яті, а доступ до них можливий за вказівником або за посиланням (відсилкою). У такій ситуації копіювання об'єкта стає нетривіальною задачею, адже просте присвоєння вказівника чи посилання копіює адресу, а не сам об'єкт. Різні мови програмування надають різні засоби для поверхневого (shallow copy) та глибокого (deep copy) копіювання. Розглянемо їх.

Як модель використаємо поліморфну колекцію об'єктів, екземплярів різних класів, що належать до однієї ієрархії. Щоб не переобтяжувати міркування структурними чи алгоритмічними складнощами, модель має бути простою і, водночас, достатньою для того, щоб показати методи подолання труднощів глибокого копіювання поліморфних об'єктів. Для цього оголосимо три класи: один базовий і два його підкласи, наприклад, ієрархію класів планіметричних фігур у складі абстрактного класу *Фігура* та класів *Прямокутник* і *Круг*. Створимо колекцію з кругів і прямокутників і покажемо, як створити її повноцінну копію. Діаграма класів моделі зображена на рис. 1.

Порівняємо можливості щодо копіювання об'єкті чотирьох популярних мов програмування високого рівня: C++, C#, Python і Pharo (сучасне втілення класичної системи Smalltalk). Щоразу оголошуватимемо ієрархію класів відповідною мовою і наповнюватимемо їхніми екземплярами деякий послідовний контейнер. Далі спробуємо створити копію цього контейнера і змінити елементи копії. А потім переконаємося, чи зазнають змін елементи оригінального контейнера.

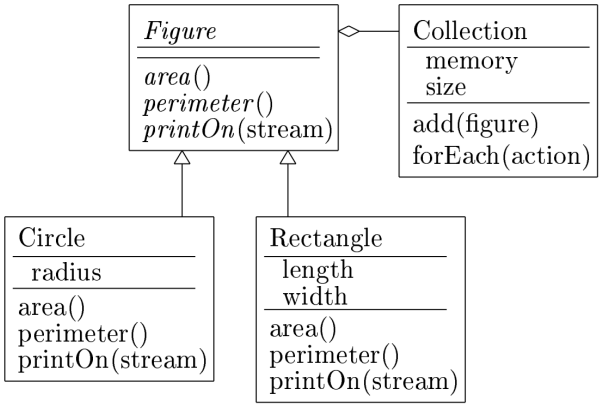

Рис. 1. Діаграма класів

2. Реалізація мовою C++

За копіювання об'єктів у програмах мовою C++ відповідають конструктор копіювання і оператор копіювального присвоєння [1]. Компілятор згенерує їх автоматично, якщо програміст не визначить у класі власні версії цих конструктора і оператора. Такі "автоматичні" члени класу забезпечують поверхневе копіювання полів об'єкта. Екземпляри проєктованих типів *Круг* і *Прямокутник* міститимуть тільки числа, тому простого копіювання значень полів цілком достатньо, і можна не турбуватися про визначення спеціальних конструкторів чи операторів присвоєння. Відкладемо цю турботу до моменту створення колекції об'єктів.

Програмний код оголошення класів у C++ прийнято ділити на дві частини: файл заголовків зазвичай містить тільки оголошення прототипів методів, а реалізацію методів розташовують у файлі вихідного коду. Заради простоти запишемо все в одному місці так, ніби методи класів визначено у файлі заголовків (тим більше, що вони доволі прості).

```
class Figure
{
public:
   virtual ~Figure() { }
   virtual double area() const = 0;
   virtual double perimeter() const = 0;
   virtual void printOn(std::ostream& os) const = 0;
   virtual Figure& scalingBy(double coef) = 0;
};
std::ostream& operator<<(std::ostream& os, const Figure& fig)
{      fig.printOn(os); return os;      }

class Circle :public Figure
{
private: double radius;
```

```
public:
   Circle(double r = 1.) :radius(r) { }
   virtual double area() const override
   {      return M_PI * radius * radius;      }
   virtual double perimeter() const override
   {      return 2.0 * M_PI * radius;      }
   virtual void printOn(std::ostream& os) const override
   {      os << "Circle(" << radius << ')';      }
   virtual Circle& scalingBy(double coef) override
   {      radius *= coef; return *this;      }
};
class Rectangle :public Figure
{
private: double length, width;
public:
   Rectangle(double a = 3., double b = 4.) :length(a), width(b) { }
   virtual double area() const override
   {      return length * width;      }
   virtual double perimeter() const override
   {      return 2.0 * (length + width);      }
   virtual void printOn(std::ostream& os) const override
   {      os << "Rectangle[" << length << 'x' << width << ']';      }
   virtual Rectangle& scalingBy(double coef) override
   {      length *= coef;  width *= coef; return *this;      }
};
```

Тут метод *Figure& scalingBy(double coef)* надає можливість змінювати розміри фігури. Саме цього нам треба, щоб перевірити, чи зміни копії об'єктів вплинуть на їхні оригінали. Зауважимо, що в підкласах його перевантажено коваріантними за типом результату методами.

Для побудови колекції фігур використаємо деякий стандартний тип контейнера, наприклад, *std::vector<T>* [2]. Типом елементів поліморфного контейнера у програмі мовою C++ мусить бути вказівник на базовий клас, тому поліморфний вектор міститиме вказівники на круги та прямокутники, розташовані в динамічній пам'яті. Клас *std::vector<T>* визначає конструктор копіювання, який правильно копіює структуру пам'яті контейнера та значення його елементів. Можна сказати, що він виконує глибоке копіювання (векторної пам'яті контейнера). Це добре працює з числовими типами, рядками, розташованими в стеку об'єктами, але зовсім не підходить для вказівників: конструктор скопіює адреси динамічних об'єктів, а не їх самих. Таким чином копія контейнера поділятиме з оригіналом один набір об'єктів.

Стандартний деструктор вектора теж виконає не все, що треба виконати у випадку, коли елементами контейнера є вказівники на динамічні об'єкти: він очистить пам'ять від внутрішніх структур вектора, але залишить у динамічній пам'яті об'єкти. Саме тому для створення поліморфних контейнерів рекомендовано використовувати "розумні" вказівники замість звичайних. Бібліотека STL пропонує на вибір такі типи: *std::unique_ptr<T>*, *std::shared_ptr<T>* і *std::weak_ptr<T>*. Перший з них забезпечує одноосібне володіння об'єктом, другий – спільне з використанням підрахунку посилань, а третій – слабке посилання на об'єкт, яким

володіє *shared_ptr*. Ми могли б використати для побудови колекції фігур *shared_ptr* або *unique_ptr*, і деструктор працював би правильно, але конструктор копіювання все ж не забезпечить бажаної поведінки. Контейнер типу *vector<shared_ptr<T>>* поділятиме об'єкти зі своєю копією, а типу *vector<unigue_ptr<T>>* – передасть об'єкти своїй копії.

Ми потребуємо іншої поведінки, тому оголосимо новий клас, підклас *std::vector<T>*. Підклас у значній мірі використовуватиме можливості базового класу. Наприклад, метод *pVector<pT>::clear()* після очищення динамічних об'єктів викличе базову реалізацію *vector::clear()*, щоб контейнер повівся, як звичайно.

Подібно конструктор копіювання *pVector<pT>* спочатку викличе конструктор базового класу, щоб скопіювати внутрішні структури вектора, а тоді в циклі спробує скопіювати кожен його елемент. Але що, власне, тут копіювати: круг, чи прямокутник? Тип елемента контейнера – вказівник на базовий тип, часто абстрактний, тому скопіювати без додаткових зусиль об'єкт, який за ним ховається, не вдасться. У C++ є оператор *typeid*, який дає змогу розпізнати тип об'єкта, але в загальному випадку (як в оголошенні нашого контейнера) він не допоможе.

Один з принципів ООП гласить: "Не питай у об'єкта, хто він, а проси виконати певну роботу". Хто найкраще знає, якого типу об'єкт? Він сам. То попросимо в елемента контейнера повернути свою копію!

```
template <typename pT>
class pVector :public vector<pT>
{
private:
   void internalClear() // очищає динамічні об'єкти
   {   for (auto it = this->begin(); it != this->end(); ++it) delete *it;
   }
public:
   pVector() : vector() { }
   ~pVector() {   internalClear();
   }
   pVector(const pVector& pV) : vector<pT>(pV) {
      // виконує глибоке копіювання
      for (auto it = begin(); it != end(); ++it) *it = (*it)->clone();
   }
   pVector(const std::initializer_list<pT>& list) {
      this->reserve(list.size()); for (auto x : list) this->push_back(x);
   }
   pVector& operator=(const pVector& pV) {
      if (this != &pV) {
         internalClear(); vector::operator=(pV);
         for (auto it = begin(); it != end(); ++it) *it = (*it)->clone();
      }
      return *this;
   }
   void clear() {
      internalClear(); vector::clear();
   }
};
```

Тут в конструкторі копіювання і в операторі копіювального присвоєння кожному елементові контейнера надсилають повідомлення *clone()*, щоб отримати у відповідь його копію.

Для того, щоб можна було використати *pVector<Figure*>*, доведеться доповнити реалізацію класів фігур методами *clone()*.

```
class Figure abstract
{
public: . . . . .
   virtual Figure* clone() const = 0;
};

class Circle :public Figure
{
private: double radius;
public: . . . . .
   virtual Circle* clone() const override
   {      return new Circle(*this);      }
};

class Rectangle :public Figure
{
private: double length, width;
public: . . . . .
   virtual Rectangle* clone() const override
   {      return new Rectangle(*this);      }
};
```

Тепер можна завершити експеримент: створити екземпляр контейнера, наповнити його фігурами, скопіювати. Легко переконатися, що зміни елементів копії ніяк не впливають на оригінал.

```
int main()
{
   pVector<Figure*> origin{ new Circle(), new Rectangle(),
      new Circle(2.), new Rectangle(2., 1.) };
   print_vector("--- The origin vector of figures", origin);
   pVector<Figure*> copy(origin);
   for (Figure* fig : copy) fig->scalingBy(2.);
   print_vector("\n--- The copy after scaling by 2.0", copy);
   print_vector("\n--- The origin after scaling copy by 2.0", origin);
}
void print_vector(const char* title, const pVector<Figure*>& vect)
{
   std::cout << title << '\n';
   for (Figure* fig : vect) std::cout << *fig
      << "\n  S = " << fig->area() << "   P = "
      << fig->perimeter() << '\n';
}
```

Повний текст програмного проєкту можна завантажити з [3].

3. Реалізація мовою C#

Відповідальність за копіювання об'єктів у програмах мовою C# розподілена інакше, ніж у C++. Тут не прийнято оголошувати конструктор копіювання. Натомість всі класи наслідують від *System.Object* [4] метод *object MemberwiseClone()*, здатний виготовити поверхневу копію довільного об'єкта. Також класи реалізують (або ні) інтерфейс *IClonable*.

```
public interface ICloneable
{
    public object Clone();
}
```

У обох випадках тут використано найзагальніший тип *object*, що не сприяє зручності та надійності використання цих засобів, адже щоразу доводиться приводити отриману копію до конкретного типу. Ліпше було б використовувати узагальнений інтерфейс *IClonable*<*T*>, але в системі такого немає.

Мова C# пропонує багату бібліотеку контейнерів. Усі вони є типами-посиланнями, тому звичайне присвоєння між змінними типу контейнера означає копіювання адрес. Зрозуміло, що в такій ситуації завдання глибокого копіювання є актуальним. Один з найпростіших контейнерів *Array* реалізує інтерфейс *IClonable*, але його метод *Clone()* діє подібно до конструктора копіювання контейнера *std::vector*<*T*> мови C++: копіює структури масиву, але не копіює елементів масиву за посиланнями на них, навіть, якщо вони реалізують *IClonable*.

Замість масиву можна використовувати контейнер *List*<*T*>, схожий за можливостями та поведінкою на *std::vector*<*T*>. Зазначимо лише, що на відміну від *Array* він не реалізує інтерфейс *IClonable*. І масив *Figure[] figures*, і список *List*<*Figure*> *list* – поліморфні. Якщо *Figure* – клас, то елементами *figures* чи *list* можуть бути екземпляри будь-яких підкласів *Figure*. Ще одна приємна властивість контейнерів C# – це підтримка автоматичного збирання сміття. Можна не турбуватися про коректне очищення динамічної пам'яті від об'єктів.

Ні масив, ні список не підтримують глибоке копіювання, то як створити копію контейнера? Можна використати спеціальні методи контейнера, якщо вони є, або загальні методи LINQ [5]. Нижче наведено приклади, як це зробити за припущення, що елементи контейнера у відповідь на повідомлення *Clone()* повертають свою копію.

```
// метод контейнера
List<Figure> list_copy = list.ConvertAll(fig => fig.Clone());
// LINQ
Figure[] arr_copy = figures.Select(fig => fig.Clone()).ToArray();
```

Після всіх цих міркувань ми готові оголосити класи фігур і написати головну програму, яка створює список фігур і експериментує з його копією. Відразу визначимо в класах метод *Clone()*, але простіше це буде зробити без використання інтерфейсу *IClonable*. Замість згаданого раніше методу *void printOn(stream)* у класах визначено *string ToString()* – це загальна практика для програм мовою C#.

```
abstract class Figure
{
    public abstract double area();
    public abstract double perimeter();
    public abstract Figure scalingBy(double coef);
    public abstract Figure Clone();
}
class Circle : Figure
{
    private double radius;
    public Circle(double r = 1.0) => radius = r;
    public override double area() => Math.PI * radius * radius;
    public override double perimeter() => 2.0 * Math.PI * radius;
    public override Circle scalingBy(double coef) {
        radius *= coef; return this;
    }
    public override Circle Clone() => new Circle(this.radius);
    public override string ToString() => $"Circle({radius})";
}
class Rectangle : Figure
{
    private double length, width;
    public Rectangle(double a= 3.0, double b = 4.0) {
        length = a; width = b;
    }
    public override double area() => length * width;
    public override double perimeter() => 2.0 * (length + width);
    public override Rectangle scalingBy(double coef) {
        length *= coef; width *= coef; return this;
    }
    public override Rectangle Clone() =>
        new Rectangle(this.length, this.width);
    public override string ToString() =>
        $"Rectangle[{length}x{width}]";
}
class Program
{
    static void Main(string[] args)
    {
        List<Figure> origin = new() { new Circle(), new Rectangle(),
            new Circle(2.0), new Rectangle(2.0, 1.0) };
        PrintList("--- The origin list of figures", origin);
        List<Figure> copy = origin.ConvertAll(book => book.Clone());
        foreach (Figure fig in copy) fig.scalingBy(2.0);
        PrintList("\n--- The copy after scaling by 2.0", copy);
        PrintList("\n--- The origin after scaling copy by 2.0",
            origin);
    }
```

```
static void PrintList(string title, List<Figure> list)
{
    Console.WriteLine(title);
    foreach (Figure fig in list) Console.WriteLine(
        \$"{fig}\n  S = {fig.area()}   P = {fig.perimeter()}");
}
```

Повний текст програмного проєкту можна завантажити з [3].

4. Реалізація мовою Python

У попередніх двох програмах абстрактні базові класи оголошували інтерфейс взаємодії з об'єктами і слугували основою для створення поліморфних колекцій. У мовах програмування зі статичною типізацією (як C++ і C#) без таких класів не обійдешся. Мови Python і Pharo підтримують динамічну типізацію. У таких програмах в одну колекцію можна поміщати різні об'єкти незалежно від їхнього типу, опрацювання повідомлень об'єкт виконує вже на етапі виконання, тому, здавалося б, можна обійтися без використання абстрактних класів. Проте і в мовах з динамічною типізацією оголошення абстрактних базових класів усіляко заохочується. Наявність спільного інтерфейсу робить програмний код читабельнішим і надійнішим, адже всі абстрактні методи потрібно перевизначити в підкласах.

На жаль, Python не надає синтаксичних засобів для оголошення абстрактного класу чи методу. З цією метою використовують модуль *abc*: абстрактний клас наслідують від *abc.ABC*, а метод роблять абстрактним за допомогою декоратора *abc.abstractmethod* [6].

Роль конструктора в класі Python відіграє "магічний" метод з іменем *__init__*. Якщо всім параметрам такого методу надати значення за замовчуванням, то він може діяти і як конструктор за замовчуванням, і як конструктор з параметрами, але аж ніяк він не може бути конструктором копіювання. У Python реалізовано загальні методи копіювання об'єктів. Їх надає модуль *copy*: функція *copy.copy(obj)* будує поверхневу копію об'єкта *obj*, а функія *copy.deepcopy(obj)* – глибоку [7]. Окремі класи мають свої методи копіювання, наприклад, *list.copy()* або *dict.copy()*. А ще в будь-якому класі можна оголосити "магічні" методи *__copy__()*, *__deepcopy__()*, щоб реалізувати особливі способи копіювання: відповідно поверхневого та глибокого.

Оголосимо ієрархію класів геометричних фігур. Закладати в них особливу поведінку щодо копіювання немає потреби. Зазначимо тільки, що в класах Python визначають метод *__str__()*, щоб об'єкти вміли перетворити себе на рядок. Про деструктори можна не турбуватися, бо в Python також діє автоматичне збирання сміття.

```
from abc import ABC, abstractmethod
class Figure(ABC):
    @abstractmethod
    def area(self):
        pass
    @abstractmethod
    def perimeter(self):
```

```
        pass
    @abstractmethod
    def scalingBy(self, coef):
        pass

class Circle(Figure):
    def __init__(self, r = 1.0):
        self.radius = r
    def __str__(self):
        return f'Circle({self.radius})'
    def area(self):
        return math.pi * self.radius ** 2
    def perimeter(self):
        return 2.0 * math.pi * self.radius
    def scalingBy(self, coef):
        self.radius *= coef
        return self

class Rectangle(Figure):
    def __init__(self, a = 3.0, b = 4.0):
        self.length = a
        self.width = b
    def __str__(self):
        return f'Rectangle[{self.length}x{self.width}]'
    def area(self):
        return self.length * self.width
    def perimeter(self):
        return 2.0 * (self.length + self.width)
    def scalingBy(self, coef):
        self.length *= coef
        self.width *= coef
        return self
```

Для побудови колекції фігур природно використати список. Це вбудований тип мови Python з широкою синтаксичною підтримкою. Копію списку *figures* дуже легко отримати за допомогою зрізу *figures[:]* або методу *figures.copy()*. Проте, в обох випадках отримаємо поверхневу копію: новий список поділятиме посилання на об'єкти з оригіналом. Щоб отримати глибоку копію, можна використати стандартну функцію *copy.deepcopy(figures)* або генератор списку, який почергово скопіює кожен елемент: *list(copy.copy(fig) for fig in figures)*. Другий спосіб надає більше контролю програмістові.

```
def printList(title, lst):
    print(title)
    for fig in lst:
        print(fig)
        print(f"  S = {fig.area()}   P = {fig.perimeter()}")

Origin = [ Circle(), Rectangle(), Circle(2.0), Rectangle(2.0, 1.0) ]
```

```
printList("--- The origin list of figures", Origin)
Copy = copy.deepcopy(Origin)
for fig in Copy:
    fig.scalingBy(2.0)
printList("\n--- The copy after scaling by 2.0", Copy)
printList("\n--- The origin after scaling copy by 2.0", Origin)
```

Як і в попередніх програмах, зміна копій ніяк не впливає на початковий список. Повний текст програмного проєкту можна завантажити з [3].

5. Реалізація мовою Pharo

Pharo – динамічно типізована мова, в якій усе є об'єктом: екземпляр класу, клас, компілятор, компільований метод тощо. Об'єкти взаємодіють через надсилання повідомлень Об'єктна модель Pharo використовує просте відкрите наслідування з одним базовим класом *Object* для всієї ієрархії. Поля даних – захищені, методи – відкриті, віртуальні [8]. Роль конструкторів виконують методи класу (адже клас – об'єкт, тому в нього є методи). Щоб створити екземпляр класу, йому зазвичай надсилають повідомлення *new*. Відповідний метод розподіляє динамічну пам'ять для екземпляра і надсилає йому повідомлення *initialize*, щоб налаштувати значення його полів даних. Таким чином пара методів *new* та *initialize* діє як конструктор за замовчуванням у C++. Селектори методів класу, призначених для створення екземплярів, зазвичай розпочинаються префіксом *new*. Приватність у Pharo влаштовано на рівні екземпляра. Це означає, що навіть клас не має доступу до змінних свого екземпляра. Для цього йому потрібні метод-селектор і метод-модифікатор. Селектори таких методів зазвичай збігаються з іменем змінної.

Використання абстрактних класів активно підтримується середовищем програмування. Щоб зробити клас абстрактним, достатньо в тілі одного його методу надіслати повідомлення *self subclassResponsibility*. У класах зазвичай перевантажують метод *printOn: aStream*, щоб екземпляри могли виводити себе в потік. У Pharo, як і в Python, діє автоматичне збирання сміття.

Копіювання об'єктів у Pharo опирається на кілька методів класу *Object*. Метод *shallowCopy* створює новий екземпляр того ж класу, що й оригінал, і копіює значення всіх полів даних, *deepCopy* копіює сам об'єкт і рекурсивно всі об'єкти, на які той посилається. Особливий спосіб копіювання визначає *copy*: він спочатку викликає *shallowCopy*, а потім – *postCopy*. Стандартний метод *postCopy* не робить нічого, його перевизначають у класі, щоб задати, для якої частини екземпляра яке копіювання потрібно виконати. Крім згаданих у класі *Object* визначено ще низку методів, наприклад *clone*, *veryDeepCopyWith: deepCopier* тощо.

```
Object << #Figure
    slots: {};
    package: 'Compare-Project'
Figure >> area
    ^ self subclassResponsibility
Figure >> perimeter
    ^ self subclassResponsibility
Figure >> scalingBy: aNumber
    ^ self subclassResponsibility
```

```
Figure << #FCircle
    slots: { #radius };
    package: 'Compare-Project'
FCircle class >> newWithRadius: aNumber
    ^ self basicNew radius: aNumber
FCircle >> initialize
    radius := 1.0
FCircle >> radius: aNumber
    radius := aNumber max: 0.1
FCircle >> area
    ^ Float pi * radius squared
FCircle >> perimeter
    ^ Float pi * radius * 2.0
FCircle >> scalingBy: aNumber
    radius := radius * aNumber
FCircle >> printOn: aStream
    aStream nextPutAll: 'Circle(';
        print: radius; nextPutAll: ')'

Figure << #FRectangle
    slots: { #length . #width };
    package: 'Compare-Project'
FRectangle class >> newWithLength: a andWidth: b
    ^ self basicNew length: a; width: b
FRectangle >> initialize
    length := 3.0. width := 4.0
FRectangle >> length: aNumber
    length := aNumber max: 0.1
FRectangle >> width: aNumber
    width := aNumber max: 0.1
FRectangle >> area
    ^ length * width
FRectangle >> perimeter
    ^ (length + width) * 2.0
FRectangle >> scalingBy: aNumber
    length := length * coef.
    width := width * coef
FRectangle >> printOn: aStream
    aStream nextPutAll: 'Rectangle['; print: length;
        nextPut: $x; print: width; nextPut: $]
```

Програмний код у Pharo пишуть виключно в методах класів, тому головну програму опишемо в класі *FRunner*. Допоміжний метод *printCollection:withTitle:* виводить колекцію на консоль системи – вікно *Transcript*, а метод *runExample* створює і копіює колекцію фігур, змінює копію та виводить усе на консоль.

```
Object << #FRunner
    slots: {};
```

```
    package: 'Compare-Project'
FRunner class >> printCollection: collection withTitle: title
    Transcript cr; show: title; cr.
    collection do: [ :fig |
        Transcript show: fig; cr;
            show: '  S = ' , fig area printString ,
            '   P = ', fig perimeter printString; cr ]
FRunner class >> runExample
    <script>
    | origin copy |
    origin := OrderedCollection withAll: { FCircle new.
        FRectangle new. (FCircle newWithRadius: 2.0).
     (FRectangle newWithLength: 2.0 andWidth: 1.0) }.
    self printCollection: origin
        withTitle: '--- The origin collection of figures'.
    copy := origin deepCopy.
    copy do: [ :fig | fig scalingBy: 2.0 ].
    self printCollection: copy
        withTitle: '--- The copy after scaling by 2.0'.
    self printCollection: origin
        withTitle: '--- The origin after scaling copy by 2.0'
```

Отримаємо результат, дуже подібний до трьох попередніх. Повний текст програми застосунку можна завантажити з [9].

6. Висновки

Підіб'ємо короткі підсумки, зазначимо типові підходи та особливості наявних засобів у кожній з мов програмування, які ми розглянули.

- Копіювання об'єкта у C++ виконує конструктор копіювання або оператор присвоєння його класу. Ці члени класу може генерувати компілятор. Тоді вони виконують поверхневе копіювання. Щоб копіювання було глибоким, потрібно самостійно визначити конструктор копіювання і оператор присвоєння. Їхні алгоритми належать до зони відповідальності програміста.
- Мова C# підтримує стандартний інтерфейс *ICloneable*. Класи, які його реалізують, зазвичай виконують поверхневе копіювання за допомогою успадкованого методу *Object.MemberwiseClone()*. Недоліком цих засобів є використання загального типу *object*. Програміст може на власний розсуд визначати у своїх класах методи *Clone()*, *DeepClone()* і використовувати їх разом з методами контейнерів або LINQ для глибокого копіювання.
- У мові Python для копіювання об'єктів визначено модуль *copy*. Його потрібно приєднувати до програми, щоб скористатися його засобами: функціями поверхневого та глибокого копіювання. Щоб додатково налаштувати дію цих функцій, програміст може оголосити в своєму класі "магічні" методи *__copy__()*, *__deepcopy__()*.
- Pharo наділяє всі об'єкти здатністю різними способами виготовляти свої копії, бо всі класи наслідують *Object*, у якому визначено відповідні методи: *shallowCopy* для поверхневого копіювання, *deepCopy* – для глибокого, *postCopy* для перевантаження і налаштування загального методу *copy*, та ще низку інших.

У статті показано, що найрізноманітніший і найзручніший арсенал засобів копіювання надає Pharo. Усі класи наділено можливостями виготовляти свою копію: поверхневу, глибоку або налаштовувану. У Python засоби копіювання однаково добре працюють зі стандартними типами і з класами користувача. Їх винесено в окремий модуль. Мови C++ і C# надають вбудовані засоби лише для поверхневого копіювання. Про глибоке копіювання доводиться турбуватися власноруч. Автори запропонували спосіб вирішення цієї проблеми на прикладі класів мовою C++. Запропоновано також спосіб удосконалення стандартного контейнера *std::vector<T>* для того, щоб він коректно працював з поліморфними вказівниками.

Список використаної літератури

1. Copy Consructors [Електронний ресурс] / Microsoft Corporation // Microsoft Docs. – 2026. – URL: https://learn.microsoft.com/en-us/cpp/cpp/copy-constructors-and-copy-assignment-operators-cpp?view=msvc-170.
2. Class vector [Електронний ресурс] / Microsoft Corporation // Microsoft Docs. – 2026. – https://learn.microsoft.com/en-us/cpp/standard-library/vector-class?view=msvc-170.
3. Глибоке копіювання об'єктів [Електронний ресурс] / Serhiy Yaroshko // Github. – 2026. – Режим доступу: https://github.com/LNUitTutor/CopyComparison.
4. Class Object [Електронний ресурс] / Microsoft Corporation // Microsoft Docs. – 2026. – https://learn.microsoft.com/en-us/dotnet/api/system.object?view=net-10.0.
5. LINQ [Електронний ресурс] / Microsoft Corporation // Microsoft Docs. – 2026. – https://learn.microsoft.com/en-us/dotnet/csharp/linq/.
6. Abstract Base Classes [Електронний ресурс] // Python Documentation. – Режим доступу: https://docs.python.org/3/library/abc.html.
7. Shallow and deep copy operations [Електронний ресурс] // Python Documentation. – Режим доступу: https://docs.python.org/3/library/copy.html.
8. Стефан Дюкас Pharo 9 на прикладах / С. Дюкас, Дж. Ракіч [та ін.]; пер. з англ. С. Ярошко. – Львів: ЛНУ ім. Івана Франка, 2022. – 270 с. [Електронне видання] – Режим доступу: http://books.pharo.org/pharo-by-example9/.
9. Глибоке копіювання об'єктів у Pharo [Електронний ресурс] / Serhiy Yaroshko // github, 2026. – Режим доступу: https://github.com/LNUitTutor/CopyComparisonPharo.

Стаття: надійшла до редколегії 03.02.2026
доопрацьована 24.02.2026
прийнята до друку 04.03.2026

TOOLS OF HIGH-LEVEL PROGRAMMING LANGUAGES FOR DEEP COPYING OBJECTS

S.A. Yaroshko, S.M. Yaroshko

Ivan Franko National University of Lviv,
1, Universytetska str., 79000, Lviv, Ukraine,
e-mail: *serhiy.yaroshko@lnu.edu.ua*

The capabilities of the C++, C#, Python, and Pharo programming languages for copying objects are compared. If an object contains at least one polymorphic reference (or pointer), the deep copy task is no longer trivial. A sequential container containing objects of different types that model geometric shapes is used as a model example. It is shown how to declare the corresponding classes in each language, fill the collection, and build a copy of it. The checks performed have shown that the resulting copy does not depend on the original.

In C++, deep copying is performed in a "manual" way: a polymorphic vector template is declared based on *std::vector*, which correctly clears memory from dynamic objects, and correctly copies the contents of the container using special methods declared in the shape classes. Copying in C# is performed using general LINQ methods, and in Python and Pharo, using standard built-in tools. It is shown that Pharo provides the programmer with the widest range of possibilities for copying objects.

In C++, an object is copied by the copy constructor or assignment operator of its class. The compiler can generate these class members. Then they perform shallow copying. To make copying deep, a programmer needs to define the copy constructor and assignment operator themselves. Their algorithms are the programmer's responsibility. The C# language supports the standard *ICloneable* interface. Classes that implement it usually perform shallow copying using the inherited *Object.MemberwiseClone()* method. The disadvantage of these tools is the use of the type *object*. The programmer can define the *Clone()* and *DeepClone()* methods in his classes at his discretion and use them with container methods or LINQ for deep copying. In Python, the *copy* module is defined for copying objects. It must be connected to the program to use its tools: shallow and deep copy functions. To further customize the behavior of these functions, the programmer can declare the "magic" *__copy__()* and *__deepcopy__()* methods in the class. Pharo gives all objects the ability to make copies of themselves in various ways, because all classes inherit from *Object*, which defines the corresponding methods: *shallowCopy* for shallow copying, *deepCopy* for deep copying, *postCopy* for overloading and customizing the general *copy* method, and others.

Key words: deep copy, shallow copy, polymorphic collection, copy constructor, interface *ICloneable*, module *copy*, method *postCopy*.