Języki i paradygmaty programowania: Laboratorium nr 3

Podstawowe paradygmaty programowania obiektowego - wprowadzenie. Dziedziczenie.

2017-2018

mgr inż. Przemysław Walkowiak
dr inż. Michał Ciesielczyk

Instrukcja

W czasie pisania programu pamiętaj o:

1. dbaniu o czytelność kodu (odpowiednie formatowanie kodu, nazewnictwo zmiennych adekwatne do ich znaczenia, komentarze),
2. dbaniu o czytelność interfejsu z użytkownikiem (w sposób jawny pytaj użytkownika jakie dane ma podać oraz opisuj wyniki, które zwracasz),
3. przed fragmentem implementującym poszczególne zadania umieść komentarz: `/*Zadanie X */` oraz wypisz na ekranie analogiczny komunikat (X jest numerem zadania): `std::cout << "Zadanie X"<< std::endl;`,
4. każde zadanie umieść w oddzielnej funkcji (w niej dopiero należy odwoływać się do zaimplementowanych funkcji i klas),
5. zaimplementuj menu wyboru zadania, a następnie wykorzystując pętle **do–while** oraz konstrukcję **switch** wykonaj odpowiedni fragment kodu,
6. w zadaniach wymagających udzielenia komentarza bądź odpowiedzi, należy umieścić go w kodzie programu (np. w postaci komentarza albo wydrukować na ekranie),
7. w zadaniach polegających na zaprojektowaniu klasy należy utworzyć jej instancję i wykorzystać zaimplementowaną funkcjonalność.

Wprowadzenie

Dziedziczenie

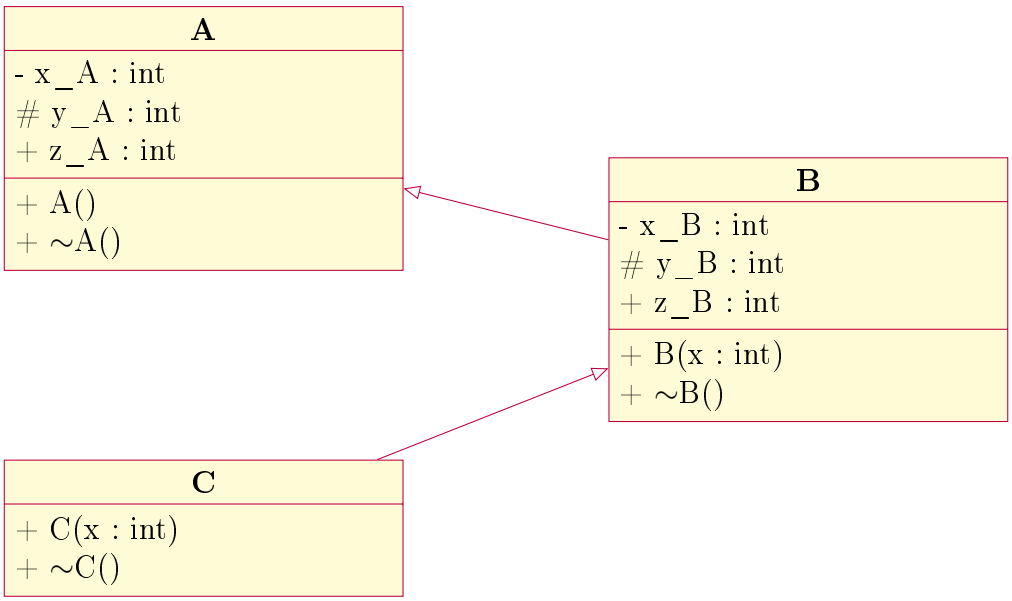
Na rysunku 1 został umieszczony diagram w notacji UML przedstawiający trzy klasy A, B oraz C. Każda klasa jest reprezentowana przez jeden prostokąt, podzielony na trzy części: nazwę klasy, listę atrybutów (pól) oraz listę metod. Symbole $-, \#, +$ przed nazwą pola lub metody oznaczają tryb widoczności danego składnika, odpowiednio: **private**, **protected** i **public**. Strzałki pomiędzy poszczególnymi klasami symbolizują relację **dziedziczenia**.

Implementacja klas A, B oraz C z rysunku 1 w języku C++ wygląda następująco:

```
class A {
    private:
        int x_A;

    protected:
        int y_A;

    public:
        int z_A;
```



Rysunek 1: Diagram zależności pomiędzy klasami A, B i C.

```
10
        A() : x_A(0), y_A(0), z_A(0) {
            cout << "ctor A" << endl;
        }

15      ~A() {
            cout << "dtor A" << endl;
        }

        void print_values_A() {
20          cout << x_A;    // ok
            cout << y_A;    // ok
            cout << z_A;    // ok
        }
};
25
class B : public A { // dziedziczenie po klasie A w trybie public
    private:
        int x_B;

30  protected:
        int y_B;

    public:
        int z_B;
35
        B(int x) : A(), x_B(x), y_B(0), z_B(0) {
            cout << "ctor B" << endl;
        }
```

```
40          ~B() {
                cout << "dtor B" << endl;
            }

            void print_values_B() {
45              // cout « x_A; // błąd: x_A jest polem prywatnym
                            // klasy bazowej A
                cout << y_A;    // ok
                cout << z_A;    // ok
                cout << x_B;    // ok
50              cout << y_B;    // ok
                cout << z_B;    // ok
            }
    }

55  class C : protected B { // dziedziczenie po klasie B w trybie
        protected
     public:
            C(int x) : B(x) {
                cout << "ctor C" << endl;
            }
60
            ~C() {
                cout << "dtor C" << endl;
            }

65          void print_values_C() {
                // cout « x_A; // błąd: x_A jest polem prywatnym
                            // klasy bazowej A
                cout << y_A;    // ok
                cout << z_A;    // ok
70              // cout « x_B; // błąd: x_B jest polem prywatnym
                            // klasy bazowej B
                cout << y_B;    // ok
                cout << z_B;    // ok
                cout << x_C;    // ok
75              cout << y_C;    // ok
                cout << z_C;    // ok
            }
    }
```

Korzystając z paradygmatu dziedziczenia w języku C++ należy pamiętać o kilku elementach:

a) ustawieniu trybu dziedziczeniu w nagłówku deklaracji klasy (linie 26 oraz 55 na powyższym listingu). Klasa `B` odziedziczy po `A` składowe z sekcji **protected** oraz **public** w niezmienionej postaci. Natomiast klasa `C` odziedziczy z `B` (oraz pośrednio z `A`) wszystkie składowe z sekcji **protected** oraz **public** jako swoje składniki sekcji **protected**.

b) wywołaniu konstruktora klasy nadrzędnej (linie 36 oraz 57). W sekcji listy inicjalizacyjnej konstruktora (lista wyrażeń pomiędzy dwukropkiem a klamrą), można jawnie wywołać odpowiedni dla nas konstruktor klasy nadrzędnej. W przypadku pominięcia jawnego wywołania, zostanie wywołany konstruktor domyślny o ile taki istnieje - jeżeli w tym konstruktor domyślny nie istnieje to w tym przypadku kompilator zgłosi błąd.

Przykłady wykorzystanie zdefiniowanych klas:

```
A a1;
B b1(5);
C c1(2);

b1.print_values_A();// wypisuje na ekranie wartości składowych
                    // x_a, y_a, z_a.
                    // Ponieważ metoda znajduje się w klasie A
                    // ma również dostęp do pól prywatnych klasy A
                    // czyli m.in. do x_a

b1.print_values_B(); // wypisuje na ekranie wartości składowych
                    // y_a, z_a oraz x_b, y_b, z_b

c1.print_values_C(); // wypisuje na ekranie wartości składowych
                    // y_a, z_a oraz y_b, z_b oraz x_c, y_c, z_c

c1.print_values_B(); //błąd: Metoda print_values_b pochodzi z klasy B
                   // po której dziedziczymy w trybie 'protected'.
                   // Dlatego do w tym miejscu
                   // nie mamy do tej metody dostępu.

A &a2 = b1;
a2.print_values_A(); // ok
a2.print_values_B(); // błąd kompilacji, klasa A nie ma tej metody.

B &b2 = c1; // błąd kompilacji. Nie można konwertować klas
            // na klasy bazowe w innych niż publicznym
            // trybie dziedziczenia.

A *a3 = &b1;
a3->print_values_A();
```

Przesłanianie metod oraz metody wirtualne

W kontekście dziedziczenia w języku C++ istnieją dwa rodzaje deklarowania metod w klasach bazowych i pochodnych:

a) metody przesłonięte,
b) metody wirtualne.

Wywołanie metody przesłoniętej jest ściśle powiązana z aktualnym typem obiektu. Rozważmy poniższy przykład:

```
class A {
    protected:
        double x, y;

    public:
        A(double x, double y) : x(x), y(y) { }

        double compute() {
            return x + y;
        }
};

class B : public A {
    public:
        B(double x, double y) : A(x,y) { }

        double compute() {
            return x * y;
        }
};
```

W liniach 8 oraz 17 zdefinowane są dwie metody o tej samej nazwie oraz liście argumentów. Dla tego przypadku metoda `compute` z linii 17 przesłania metodę z linii 8. W momencie wywołania metody `compute` wywołana zostanie zawsze ta wersja, która należy do klasy obiektu dla którego jest wywoływana. Tzn.:

```
A a1(5, 4);
cout << a1.compute(); // zostanie wywołana metoda A::compute();
// wynik: 9

B b1(5, 4);
cout << b1.compute(); // zostanie wywołana metoda B::compute();
// wynik: 20

A& a2_b1 = b1;
cout << a2_b1.compute(); // zostanie wywołana metoda A::compute();
// wynik: 9

A* a3_b1 = &b1;
cout << a3_b1->compute(); // zostanie wywołana metoda A::compute();
// wynik: 9
```

Metody przesłoniętę są zawsze przywiązane do **typu klasy**, pod jaką widoczny jest nasz obiekt. Metody wirtualne definiuje się poprzeez dodanie słowa kluczowego **virtual** przed nagłówkiem metody i w przeciwieństwie do metod przesłoniętych są one na stałe przypisane do **instancji klasy** (czyli obiektu), a nie klasy. Rozważmy analogiczny do poprzedniego przykład:

```
class A {
    protected:
        double x, y;

    public:
        A(double x, double y) : x(x), y(y) { }

        virtual double compute() {
            return x + y;
        }
};

class B : public A {
    public:
        B(double x, double y) : A(x,y) { }

        virtual double compute() {
            return x * y;
        }
};
```

W liniach 8 oraz 17 dodatkowo zostało dopisane słowo kluczowe **virtual**, które zmienia tryb w którym metody sa dziedziczone. W tym przypadku podczas zarzutowania instancji klasy B na klasę A i wywołaniu metody `compute` zostanie wywołana wersja przynależna do klasy B.

```
A a1(5, 4);
cout << a1.compute(); // zostanie wywołana metoda A::compute();
// wynik: 9
B b1(5, 4);
cout << b1.compute(); // zostanie wywołana metoda B::compute();
// wynik: 20

A& a2_b1 = b1;
cout << a2_b1.compute(); // zostanie wywołana metoda B::compute();
// wynik: 20

A* a3_b1 = &b1;
cout << a3_b1->compute(); // zostanie wywołana metoda B::compute();
// wynik: 20
```

Wirtualny destruktor

W języku C++ oprócz metod wirtualnymi mogą być również destruktory (konstruktory nie mogą być wirtualne). Wykorzystywane jedyną i najważniejszą różnicą w stosunku do klasycznych destruktorów jest to, że są one wywoływane również w momencie usuwania za pomocą operatora **delete** obiektu zarzutowanego na klasę wyżej w hierarchi.

```
class A {
    A() { }
    virtual ~A() { }
}
class B : public A{
    B() { }
    virtual ~B() { }
}


B* pb = new B(); // tworzymy dynamicznie instancję klasy B
A* pa_b = pb;   // rzutujemy instancję klasy B na klasę A
delete pa_b;    // usuwamy obiekt wskazywany przez wskaźnik
```

W przypadku destruktorów wirtualnych w momencie wykonania instrukcji **delete** `pa_b` zostanie wykony zarówno desktruktor `A::~A()` jak i `B::~B()`. Gdyby pominięto słowo kluczowe **virtual**, zostałby wywołany tylko destruktor `A::~A()`, co mogłoby doprowadzić do nieprawidłowego zwolnienia zasobów przez dynamicznie utworzony obiekt.

Zadania

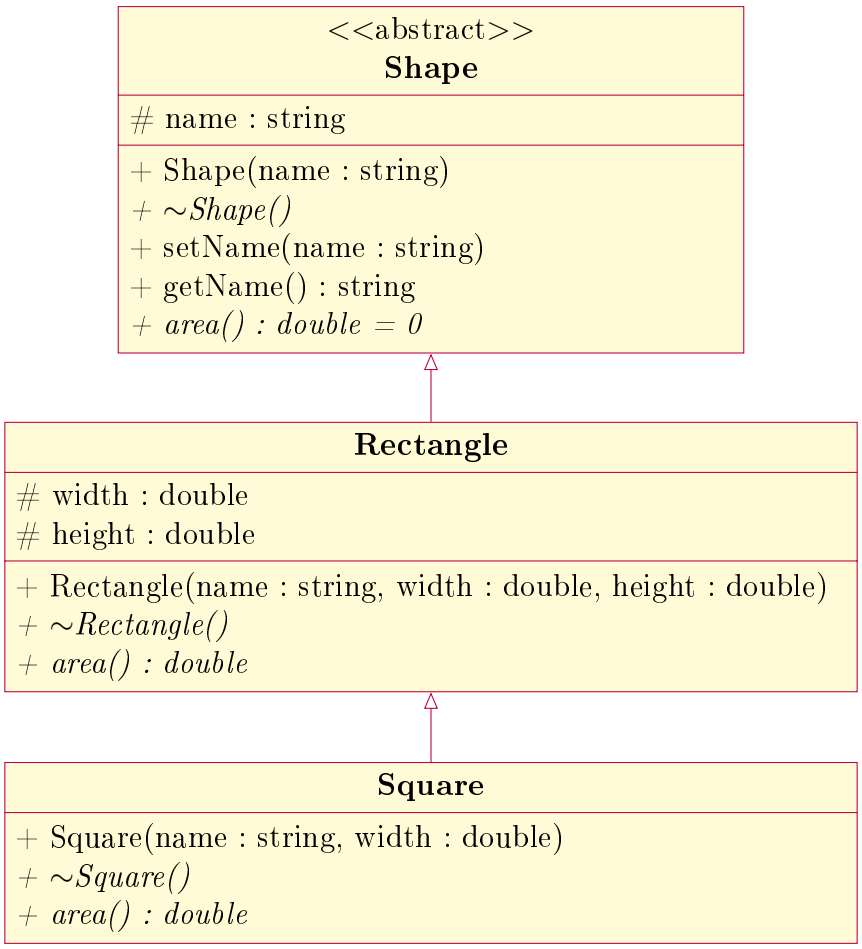
Zadanie 1

Zaimplementuj klasy `Shape`, `Rectangle` oraz `Square` ze składowymi jak na rysunku 2 i następującą funkcjonalnością:

a) każdy konstruktor i destruktor wypisuje na ekranie komunikat, że jest wywoływany,
b) w każdej klasie pochodnej zaimplementuj metodę `area()` liczącą pole figury (w klasie `Shape` powinna to być metoda czysto wirtualna).

Następnie wykorzystaj zaimplementowane klasy w różnych wariantach:

a) utwórz instancję każdej klasy i wyświetl na ekranie pole figur,
b) z wykorzystaniem referencji zarzutuj obiekty każdej z klas `Rectangle` i `Square` na wszystkie klasy wyżej w hierarchii oraz wywołaj metodę `area()`. Co zauważyłeś/aś?
c) z wykorzystaniem wskaźników zarzutuj obiekty każdej z klas `Rectangle` i `Square` na wszystkie klasy wyżej w hierarchii oraz wywołaj metodę `area()`. Co zauważyłeś/aś?
d) oznacz metodę `area()` jako metodę wirtualną i wykonaj ponownie warianty b i c,
e) zmień tryb dziedziczenia klasy `Square` na **protected** i spróbuj wykonać ponownie warianty b i c. Co zauważyłeś/aś?



Rysunek 2: Diagram zależności pomiędzy klasami Shape, Rectangle, Square.

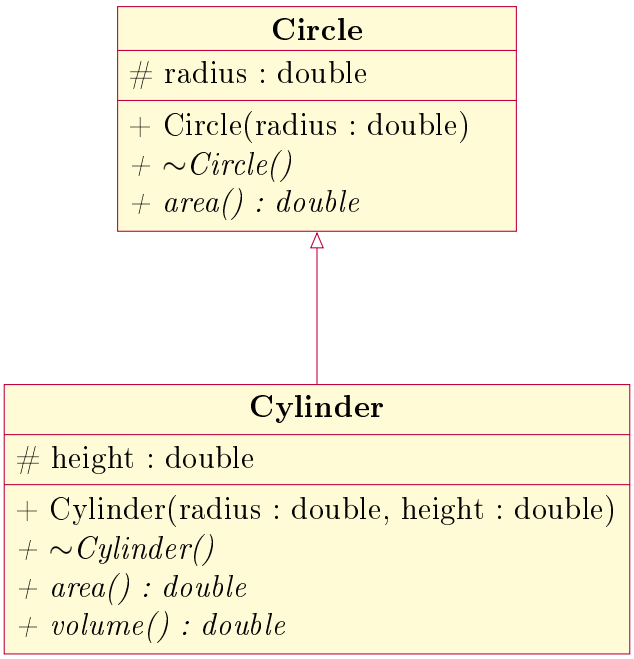
Wykonaj powyższe polecenia (wnioski napisz w komentarzu). Prześledź w jakiej kolejności dla obiektów automatycznych wywoływane są konstruktory, a w jakiej destruktory.

Wskazówka 1 W konstruktorach klas dziedziczących wywołaj konstruktor klasy bazowej. Przykładowo, implementacja konstruktora `Square` mogłaby wyglądać następująco:

```
Square(std::string name, double width) :
    Rectangle(name, width, width) // wywołanie konstruktora Rectangle
    {}
```

Zadanie 2

Zaimplementuj klasę reprezentującą bryłę w kształcie walca rozszerzającą klasę reprezentującą koło z poprzedniego zadania oraz pozwalającą na wyznaczenie objętości oraz pola powierzchni tej bryły. Przykładowy diagram klas przedstawiono na rysunku 3. W celu wyznaczenia objętości bryły wykorzystaj metodę wyznaczającą pole powierzchni w klasie bazowej.



Rysunek 3: Przykładowy diagram klas dla Circle oraz Cylinder.

Zadanie 3*

Przygotuj interfejs `Funkcja` (klasa posiadająca wyłącznie funkcje wirtualne) zawierający konstruktor bezargumentowy dostępny wyłącznie dla klas dziedziczących oraz jedną metodę czysto wirtualną `oblicz(float x)`. W klasach pochodnych metoda `oblicz(float x)` powinna zwracać wartość funkcji przechowywanej w obiekcie w zadanym punkcie `x`.

Następnie, zaimplementuj klasę `FunkcjaLiniowa` będącą pochodną klasy `Funkcja`. Klasa `FunkcjaLiniowa` powinna zawierać pola `a` i `b` oraz przeciążoną metodę `oblicz(float x)` w taki sposób, żeby zwracała wartość funkcji $a * x + b$.

Zadanie 4*

Zaimplementuj funkcję `bisekcja`, która otrzymuje jako argumenty wskaźnik do obiektu klasy pochodnej klasy `Funkcja` z poprzedniego zadania, liczby p, k oraz d i szuka miejsca zerowego przekazanej w argumencie funkcji metodą bisekcji w przedziale od p do k. Funkcja ma zwrócić miejsce zerowe z dokładnością do d. Jeżeli wartości funkcji na końcach zadanego przedziału są tego samego znaku to funkcja może zwrócić cokolwiek.

Dodatkowe informacje:

- Metoda bisekcji - http://pl.wikipedia.org/wiki/Metoda_równego_podziału