Języki programowania obiektowego

Laboratorium IV

Klasy abstrakcyjne i interfejsy, Metoda wytwórcza, Zasady segregacji interfejsów

Klasy abstrakcyjne

Klasy abstrakcyjne to, najogólniej, takie klasy dla których nie można utworzyć obiektu. Można by zadać pytanie: "Po co więc nam taka klasa?". Odpowiedź jest prosta: aby z niej dziedziczyć. Rozpatrzmy następujący przykład. Często zdarza się że mamy kilka klas które mają pewną ilość cech wspólnych, aczkolwiek między nimi samymi nie zachodzi relacja dziedziczenia (żadna z klas nie jest szczególnym przypadkiem innej klasy). Przykład z życia wzięty to na przykład zwierzęta: `Pies`, `Kot`, `Mysz`. Nie mamy tu żadnej relacji dziedziczenia (ani tym bardziej kompozycji), ale można zauważyć że wszystkie trzy mają pewne cechy wspólne (każde ma pewną ilość nóg, kolor sierści, wagę, wielkość, wydawane odgłosy itd). Moglibyśmy więc wydzielić bazową klasę `Zwierze` gdzie zawarliśmy te wszystkie wspólne cechy. Daje nam to szereg korzyści:

- jeśli postanowimy zmienić jakieś pole wspólne, to zmiany dokonujemy tylko w klasie bazowej,
- jeśli pojawią się nowe wspólne cechy które potrzebujemy to dodajemy je tylko w jednej klasie (te dwie cechy chronią nas przed popełnieniem błędu)
- chyba najważniejsza kwestia - możliwość wywołań polimorficznych

Nie trudno zauważyć że większość "funkcji" psa, kota czy myszy się pokrywa. Każde z nich je, śpi, biega, bawi się, wydaje dźwięki itd. Każde z nich robi to w inny sposób, ale jednak funkcja jako taka jest ta sama. Wspólne funkcjonalności dla każdej klasy potomnej najlepiej jest zaimplementować w klasie bazowej jako funkcje wirtualne. W naszym przykładzie będzie to wyglądać w ten sposób, że w klasie bazowej `Zwierze` utworzymy na przykład funkcję wirtualną `sleep()` a następnie w klasach potomnych nadpiszemy tą bazową funkcję wirtualną. Dla przypomnienia, nadpisanie funkcji bazowej oznacza, że w klasie pochodnej tworzymy metodę o tej samej nazwie oraz argumentach i wartości zwracanej co w klasie bazowej. W klasie potomnej metoda ta implementuje działania odpowiednie dla swojej podklasy. W klasycznym dziedziczeniu między "zwykłymi" klasami funkcje wirtualne zawsze miały pewne definicje zarówno w klasie bazowej jak i potomnych. To znaczy jeśli mieliśmy np klasy: `ZegarekNaReke` i `ZlotyZegarekNaReke` to mogliśmy wywołać metodę `podajCzas()` dla obiektów obu tych klas. W klasach abstrakcyjnych, które nie mogą mieć obiektów, metody wirtualne nie muszą mieć tych definicji. Aby klasa była abstrakcyjna musi mieć przynajmniej jedną metodę czysto wirtualną – czyli metodę wirtualną która nie ma ciała**.** Na przykład:

```
class A
{
 public:
     virtual void metodaCzystoWirtualna() = 0; // metoda czysto wirtualna
```

```
};
```

Każda klasa z metodą czysto wirtualną będzie abstrakcyjna. Każda klasa dziedzicząca z takiej klasy także, dopóki nie będzie w niej definicji wszystkich metod czysto wirtualnych. Definiowanie takich metod w klasach pochodnych nie różni się niczym od definiowania innych metod.

```
class B : public A
{
 public:
      virtual void metodaCzystoWirtualna() {} //już nie jest czysto wirtualna, bo ma ciało - choćby i puste
};
```

Oczywiście próba utworzenia obiektu klasy abstrakcyjnej skończy się błędem kompilatora.

Rozważmy jeszcze inny przykład zastosowania klas abstrakcyjnych. Stwórzmy hierarchię klas odpowiedzialną za przechowywanie kolejki zmiennych typu `int`. Załóżmy że chcemy mieć dwa typy kolejek. `ArrayQueue` (kolejka wykorzystująca tablicę) i `ListQueue` (kolejka wykorzystująca listę). Nie trudno zauważyć, że możemy wyodrębnić dla tych klas pewną wspólną bazę. Obie kolejki potrzebują metod: `push()`, `pop()`, `size()`, `remove()`, `exists()`, `search()`. Może nam się też przydać pole `iloscElementow`. Widać też wyraźnie, że tworzenie obiektu klasy `Kolejka` nie miałoby sensu, gdyż sama `Kolejka` nie ma jednoznacznie zdefiniowanego sposobu przechowywania danych. Dlatego możemy zdefiniować abstrakcyjną klasę `Queue` zawierającą metody czysto wirtualne i dla której nie będzie można utworzyć instancji. Na przykład klasa `Queue` mogłaby wyglądać następująco:

```
class Queue
{
 protected:
      int iloscElementow;
 public:
      virtual void push(int v) = 0;
      virtual int pop() = 0;
};
```

Klasy dziedziczące z klasy `Queue` muszą mieć już zdefiniowany sposób przechowywania elementów, na przykład `ArrayQueue` musi mieć dodatkowe pole które będzie tablicą (lub wskaźnikiem do jej pierwszego elementu) z danymi.

Jeśli w trakcie rozwoju programu okaże się, że będziemy potrzebowali w naszych kolejkach (a będzie ich na przykład 10 rodzajów) dodatkowe pole to dodanie go sprowadzi się do modyfikacji jedynie naszej klasy abstrakcyjnej. Wyodrębnienie klas abstrakcyjnych bardzo pomaga przy bardziej złożonych hierarchiach klas. Dekomponowanie problemu na mniejsze fragmenty pozwala skupić się na mniejszej ilości spraw i uniknąć błędów. Dodatkowo możemy przetestować "małe" klasy w których łatwiej szuka się błędów. Oczywiście nie warto przesadzać w drugą stronę – tworzenie nadmiernej ilości klas potomnych może również utrudnić zarządzanie i rozwój projektu.

Interfejsy

Interfejs to klasa abstrakcyjna która ma tylko i wyłącznie metody czysto wirtualne i nie ma żadnych pól. Interfejsy są bardzo przydatne gdy chcemy zupełnie niezwiązanym ze sobą obiektom udostępnić taki sam zestaw metod. Rozpatrzmy następujący przykład. Załóżmy że chcemy napisać funkcję `min(a, b)`, która będzie zwracała mniejszą wartość spośród dwóch argumentów. Wiadomo, że aby ta funkcja mogła działać, potrzebuje otrzymać dwa obiekty które można w jakiś sposób porównać (istnieje jakaś relacja porządku w zbiorze tych obiektów). Powstaje pytanie w jaki sposób zagwarantować że nasza funkcja dostanie właśnie takie obiekty? Co więcej, jakiego właściwie typu argumenty powinna posiadać? Rozwiązaniem jest stworzenie interfejsu który nazwiemy `IComparable`.

Interfejs ten będzie zawierał wirtualną funkcję operatorową `operator<`, która będzie służyła do porównania dwóch obiektów, które implementują (dziedziczą) ten interfejs. Na przykład:

```
class IComparable
{
 public:
    virtual bool operator<(const IComparable&) const = 0;
};
```

Dobrą praktyką jest by nazwy klas abstrakcyjnych będących interfejsami zaczynać od dużej litery `I`. Umożliwia to od razu rozpoznanie, że dana klasa jest interfejsem. Klasy potomne, które będą dziedziczyć z tego interfejsu, będą nadpisywały (implementowały) metodę `operator<`. W klasie potomnej metoda `operator<` będzie wykonywać porównanie dwóch obiektów w odpowiedni dla swojego typu sposób. Na przykład, klasa `Box` reprezentująca klocek oraz implementująca interfejs `IComparable` może wyglądać następująco:

```
class Box : public IComparable
{
 private:
     int m_weight;
 public:
     Box(int weight) : m_weight(weight) {}

     virtual bool operator<(const IComparable& rhs) const
     {
        const Box& b = dynamic_cast<const Box&>(rhs);
        return m_weight < b.m_weight;
     }

     friend ostream& operator<<(ostream& out, const Box& b)
     {
       out << b.m_weight;
       return out;
     }
};
```

W klasie Box implementujemy funkcję operatorową `operator<(const IComparable& rhs)` odziedziczoną z interfejsu `IComparable`, która w tym przypadku porównuje dwa klocki na podstawie ich wagi. Należy zwrócić uwagę na linię `const Box& b=dynamic_cast<const Box&>(rhs);`. Argument `rhs` jest stałą referencją do klasy `IComparable`. Ponieważ jest to klasa nadrzędna dlatego nie widzi ona pola `m_weight`, które jest wprowadzone w klasie pochodnej `Box` i które jest nam potrzebne do wykonania porównania. Zatem aby odwołać się do pola `m_weight` za pomocą argumentu `rhs`, musimy go *explicite* rzutować na stałą referencję do klasy potomnej za pomocą operatora `dynamic_cast`. Istnieje jednak pewne niebezpieczeństwo polegające na tym, że jako drugi argument operatora (`rhs`) może zostać przekazany obiekt innej klasy, która wprawdzie implementuje interfejs `IComparable` ale nie posiada pola `m_weight`. W takim przypadku operator `dynamic_cast` zgłosi wyjątek `std::bad_cast`.

Ostatecznie funkcja `min(a, b)` zwracająca mniejszy z dwóch argumentów będzie mieć postać:

```
const IComparable& min(const IComparable& a, const IComparable& b)
{
  if ( a < b )
    return a;
  else
    return b;
}
```

Teraz jeśli stworzymy dwa klocki to możemy już je porównać za pomocą operatora `<`.

```
int main()
{
	Box b1(10);
	Box b2(12);
	Box b = min(b1,b2);
	cout << b << endl;

	return 0;
}
```

Zasada segregacji interfejsów

Zasada segregacji interfejsów jest kolejną zasadą SOLID, która głosi, że nie należy tworzyć interfejsów z metodami, których nie używa konkretna klasa ten interfejs. Oznacza to, że interfejsy powinny być jak najmniejsze i konkretne klasy nie powinny implementować metod których nie potrzebują. Nie powinno dojść do sytuacji, gdy któraś z klas pochodnych nie wykorzystuje zaimplementowanej w interfejsie metody. Wyobraźmy sobie, że mamy interfejs, który jest używany w kilkunastu innych projektach, jednak w każdym projekcie używana jest tylko jedna metoda tego interfejsu. Gdy zajdzie potrzeba zmiany tego interfejsu spotkamy się z problemem ingerowania w każdy projekt. Nie powinniśmy zmuszać klasy do implementowania metod, których nie potrzebuje. Zgodnie z zasadą segregacji interfejsów należy zdefiniować większą liczbę małych i lekkich interfejsów. Rozważmy zatem następujący przykład.

```
class Car {
public:
	virtual void engineOn() = 0;
	virtual void move() = 0;
	virtual void automaticAirConditioning() = 0;
	virtual void cruiseControl() = 0;
};

class Fiat126p : public Car {
public:
	void engineOn() override {
	// turn on engine
	}
	void move() override {
	// move this car
	}
	void automaticAirConditioning() override {}
	void cruiseControl() override {}
};

class AudiA7 : public Car {
public:
	void engineOn() override {
	// turn on engine
	}
	void move() override {
	// move this car
	}
	void automaticAirConditioning() override {
	// temperature is too high, turn on air conditioning please
	}
	void cruiseControl() override {
	// we are on highway, turn on cruise control
	}
};
```

W powyższym przykładzie mamy klasy `Fiat126p` oraz `AudiA7`, które implementują interfejs `Car`. O ile Audi posiada i korzysta ze wszystkich funkcji, o tyle przypomnijmy sobie, czy Maluch korzystał z takich udogodnień jak klimatyzacja automatyczna czy tempomat? Może zdarzyły się jakieś pojedyncze przypadki, ale pomińmy je i załóżmy, że nie. Zatem klasa `Fiat126p` nie potrzebuje implementować metod z których nie będzie korzystać. Klasa `Fiat126p` powinna implementować tylko te z których korzysta. Zgodnie z zasadą segregacji interfejsów należy po prostu rozdzielić większy interfejs na mniejsze, co obrazuje poniższy przykład.

```
class Car {
public:
      virtual void engineOn() = 0;
      virtual void move() = 0;
};

class AdditionalFunctions {
public:
      virtual void automaticAirConditioning() = 0;
      virtual void cruiseControl() = 0;
};

class Fiat126p : public Car {
public:
      void engineOn() override {
      //turn on engine
      }
      void move() override {
      //move this car
      }
};

class AudiA7 : public Car, AdditionalFunctions {
public:
      void engineOn() override {
      //turn on engine
      }
      void move() override {
      //move this car
      }
      void automaticAirConditioning() override {
      //temperature is too high, turn on air conditioning please
      }
      void cruiseControl() override {
      //we are on highway, turn on cruise control
      }
};
```

Dzięki powyższemu rozwiązaniu w każdej konkretnej klasie implementujemy tylko te metody, które są nam niezbędne. Rozdzielenie interfejsów na mniejsze zapewnia nam dużo większą elastyczność. Wydzielamy osobne funkcjonalności do osobnych interfejsów a klasy implementują tylko te interfejsy, które są potrzebne. Unikamy wymuszania implementacji metod, które w danej klasie nie są używane.

Zadania

Zadanie 1

1. Stwórz hierarchię klas dla figur geometrycznych. Nadrzędna klasa `Shape` będzie klasą abstrakcyjną i powinna zawierać:
    a. chronione pole typu `string` reprezentujące kolor figury jako napis (np. “red”),

   b. metodę czysto wirtualną `float get_area()`, która zwraca pole powierzchni figury.

2. Stwórz interfejs `IMovable`, który reprezentuje obiekty mogące się poruszać na płaszczyźnie. Interfejs powinien posiadać metody:
   a. `void move(float dx, float dy)`, która przesuwa figurę o wektor `[dx, dy]`,
   b. `void move_to_origin()`, która przesuwa środek figury do początku układu współrzędnych.

3. Stwórz interfejs `IPrintable`, który reprezentuje obiekty, których stan można wypisać na ekranie. Interfejs powinien posiadać metody:
   a. `void print()`, która wypisuje na ekranie informacje o figurze.

4. Stwórz klasy `Circle` oraz `Rectangle`, które dziedziczą z klasy abstrakcyjnej `Shape` i reprezentują odpowiednio okrąg oraz prostokąt na płaszczyźnie. Dodatkowo klasy `Circle` oraz `Rectangle` implementują interfejsy `IMovable` oraz `IPrintable`. Zastanów się jakie dodatkowe pola powinna posiadać każda z tych klas oraz utwórz odpowiedni konstruktor oraz gettery i settery. Nie zapomnij o odpowiedniej implementacji wirtualnych metod odziedziczonych z klasy abstrakcyjnej `Shape` i interfejsów `IMovable` oraz `IPrintable`.

5. W funkcji `main()` napisz krótki program demonstrujący użycie klas `Circle` i `Rectangle`. Zademonstruj, działanie metod odziedziczonych po klasie `Shape` oraz użytych interfejsach.

Zadanie 2

1. Tym razem stwórz interfejs o nazwie `IVehicle`, który będzie zawierał metody reprezentujące czynności wspólne dla wszystkich pojazdów. Następnie utwórz hierarchię klas reprezentujących pojazdy różnych typów. Klasy reprezentujące konkretne pojazdy powinny implementować wirtualne metody interfejsu `IVehicle`. Napisz, krótki program demonstrujący użycie utworzonych klas.

Zadanie 3

Zmodyfikuj poniższy kod tak aby był zgodny z zasadą segregacji interfejsów.

```
class TextSaver {
public:
      void save(const string& filetype) {
      if (filetype == "doc") {
            // save text to .doc
      }
      else if (filetype == "txt") {
            // save text to .txt
      }
      }
};
```