Programowanie obiektowe

Laboratorium II

Dziedziczenie klas, zasada podstawiania Liskov

Zasada podstawiania Liskov

Zasada podstawienia Liskov (ang. *Liskov Substitution Principle*) została sformułowana po raz pierwszy przez Barbarę Liskov w książce *Data Abstraction and Hierarchy*. Zasada ta mówi o tym, że klasa dziedzicząca powinna tylko rozszerzać możliwości klasy bazowej a nie modyfikować sposób jej działania. Innymi słowy, klasa potomna nie powinna zmieniać tego, jak działa klasa nadrzędna. Oznacza to, że jeśli będziemy tworzyć egzemplarz klasy potomnej, wywoływanie metody, którą pierwotnie zdefiniowano w klasie bazowej, powinno dać te same rezultaty jak w przypadku klasy nadrzędnej. Zasada LSP dotyczy prawidłowo zaprojektowanego dziedziczenia. Jeżeli tworzymy klasę pochodną, to musimy być również w stanie użyć jej zamiast klasy bazowej. W przeciwnym przypadku oznacza to, że dziedziczenie zostało zaimplementowane nieprawidłowo. Celem zasady podstawiania Liskov jest uniknięcie sytuacji niezamierzonego lub nieprawidłowego działania stworzonego oprogramowania.

Rozpatrzmy parę przykładów. Mamy klasę `Vehicle`. Posiada ona dwie metody `turnOnEngine()` – uruchomienie silnika oraz `move()` – jedziemy. Mamy następnie dwie klasy `Car` i `Bike`, które dziedziczą po klasie `Vehicle`. Jak widać jest to idealna relacja z realnego świata, gdyż samochód jak i rower są pojazdami. Jednakże z punktu widzenia działania programu jest to błędna implementacja hierarchii klas. Jest tak dlatego, ponieważ w przypadku gdy użyjemy roweru (obiekt klasy `Bike`) to mamy możliwość uruchomienia silnika, co jest błędem projektowym, gdyż rower nie posiada silnika. Hierarchia klas, która wprawdzie ma sens w realnym świecie gdyż rower i samochód są pojazdami, w przypadku programu jest błędna gdyż powoduje, że klasa `Bike` dziedziczy metodę `turnOnEngine()`, która w jej przypadku nie ma zastosowania. Jest to naruszenie zasady podstawiania Liskov. Jak zatem poprawnie zaimplementować tą hierarchię tak aby spełniała ona zasadaę podstawiania Liskov. Należy stworzyć dodatkowe klasy `VehicleWithEngine` i `VehicleWithoutEngine`, które dziedziczą z klasy `Vehicle`. Metoda `move()` pozostanie w klasie `Vehicle` gdyż jest ona wspólna dla wszystkich pojazdów zaś metoda `turnOnEngine()` zostanie przeniesiona do klasy `VehicleWithEngine`. Klasa `Bike` będzie dziedziczyć po klasie `VehicleWithoutEngine` zaś klasa `Car` po klasie `VehicleWithEngine`. Ta hierarchia spełnia już zasadę podstawiania Liskov.

Teraz, przejdźmy do kolejnego przykładu.. Mamy następującą hierarchię klas:

```
class Vehicle
{
      public: virtual void turn_on(int temp)
      {
      if (temp < -20)
```

```
            return;

      /* … */
      }
}

class Car : public Vehicle
{
      public: void turn_on(int temp) override
      {
      if (temp < -5)
            return;

      /* … */
      }
}
```

W powyższym przykładzie, klasa pochodna `Car` jest bardziej restrykcyjna niż jej typ bazowy `Vehicle`. Klasa `Car` wymaga, aby argument `temp` była większy niż -5, podczas gdy klasa bazowa wymaga, aby argument był jedynie większy niż -20. Oznacza to na przykład, że jeśli jako argument `temp` zostanie przekazana wartość, na przykład -10, to dla tej wartości metoda `TurnOn()` z klasy nadrzędnej wykona poprawnie swoje działania, zaś metoda `TurnOn()` z klasy podrzędnej natychmiast zakończy swoje działanie. Jest przykład naruszenia zasady podstawiania Liskov. W kontekście dziedziczenia klas posługujemy się pojęciem warunki wstępne (ang. *preconditions*), które oznacza założenia dotyczące argumentów przekazywanych do funkcji. Aby hierarchia klas spełniała zasadę podstawiania Liskov musi być spełniony warunek polegający na tym, że warunki wstępne nie mogą zostać wzmocnione w podklasie - mogą natomiast zostać rozluźnione. Wzmocnienie warunków wstępnych w podklasie oznacza, że podklasa nie może wykonać tego wszystkiego co może zrobić nadklasa co oznacza naruszenie zasady podstawiania Liskov.

Kolejny podobny przykład. Mamy następującą hierarchię klas:

```
class Vehicle
{
public:
      int femp;

      virtual int get_temp()
      {
        femp = -1;
        if (femp < -100)
            throw std::runtime_error("Sensor damaged.");
        return femp;
      }
};

class Car : public Vehicle
{
public:
      int get_temp() override
      {
        femp = -200;
        return femp;
      }
};
```

W powyższym kodzie należy zwrócić uwagę na to, że metoda `get_temp()` z klasy bazowej `Vehicle` wyrzuci wyjątek, gdy `ftemp` < -100. Aby była spełniona zasada podstawiania Liskov również przynajmniej taki sam warunek powinien być zaimplementowany w metodzie `get_temp()` w klasie pochodnej `Car`. W kontekście dziedziczenia

klas posługujemy się również pojęciem warunki końcowe (ang. *postconditions*), które wiąże się z warunkami dotyczącymi wartości zwracanych przez funkcję lub ze stanem obiektu po wykonaniu funkcji. Zasada podstawiania Liskov wymaga aby warunki końcowe nie zostały rozluźnione w podklasie - mogą natomiast zostać wzmocnione. W powyższym przykładzie metoda `get_temp()` z klasy `Car` nie sprawdza wcale właściwości `ftemp`, przez co jest mniej restrykcyjna niż klasa bazowa `Vehicle`. Jest to również naruszenie zasady podstawiania Liskov gdyż warunek końcowy został rozluźniony w podlasie.

Na koniec, załóżmy teraz, że tworzymy aplikację, która wykonuje pewne operacje na figurach geometrycznych. W pierwszej wersji programu został zdefiniowany typ *Rectangle*, który reprezentuje prostokąt:

```
class Rectangle
{
  private:
      float fwidth;
      float fheight;
  public:
      void set_width(float width)
      {
        fwidth = width;
      }
      float get_width()
      {
        fwidth;
      }
      void set_height(float height)
      {
        fheight = height;
      }
      float get_height()
      {
        return fheight;
      }
};
```

Następnie, w programie mamy funkcję, która oblicza pole prostokąta:

```
float calculate_area(Rectangle* rectangle) {
  return rectangle->get_width() * rectangle->get_height();
}
```

W funkcji `main()` możemy utworzyć obiekt klasy `Rectangle`, następnie ustawić jego wymiary a na końcu obliczyć jego pole:

```
int main() {
      Rectangle rect;
      rect.set_width(5);
      rect.set_height(4);
      int area = calculate_area(&rect);
      return 0;
}
```

Funkcja `calculate_area()` w tym wypadku zwróci wynik 20.

Po pewnym czasie powstaje konieczność wprowadzenia nowej figury, na przykład kwadratu. Wiemy, że z matematycznego punktu widzenia kwadrat jest prostokątem. Zatem może się wydawać dobrym pomysłem rozszerzenie klasy `Rectangle` w następujący sposób:

```
class Square : public Rectangle
{
public:
```

```
    void set_width(float width)
    {
      Rectangle::set_width(width); // wywołujemy metodę set_width z nadklasy
      Rectangle::set_height(width);
    }

    void set_height(float height)
    {
      Rectangle::set_height(height);
      Rectangle::set_width(height);
    }
};
```

W związku z tym, że szerokość i wysokość kwadratu są równe, metody klasy bazowej `Rectangle` zostały nadpisane w taki sposób, żeby ustawienie dowolnego parametru zmieniało też drugi. Użycie nowej figury może wyglądać następująco:

```
int main() {
      //Rectangle fig;
      Square fig;
      fig.set_width(5);
      fig.setHeight(4);
      int area = calculateArea(&fig);
      return 0;
}
```

W powyższym przykładzie zwróćmy uwagę na dwa fakty. Po pierwsze, do funkcji `calculate_area()` możemy przekazać zarówno wskaźnik do obiektu klasy `Rectangle` oraz `Square` ponieważ klasa `Square` dziedziczy z klasy `Rectangle`. Po drugie, jeśli w powyższym przykładzie utworzymy obiekt klasy `Rectangle` to funkcja `calculate_area()` zwróci wartość 20, zaś jeśli utworzymy obiekt klasy `Square` to funkcja `calculate_area()` zwróci wartość 16 gdyż w tym przypadku metoda `set_height()` ustawi zarówno wysokość jak i szerokość kwadratu na 4. Jest to przykład naruszenia zasady podstawiania Liskov gdyż obiekt klasy `Square` mimo iż dziedziczy z klasy `Rectangle` to inaczej się od niej zachowuje.

Powyższy przykład obrazuje, że chociaż z matematycznego punktu widzenia kwadrat jest bardziej szczególnym przypadkiem prostokąta to jednak z programistycznego punktu widzenia kwadrat niekoniecznie jest bardziej szczegółowym przypadkiem prostokąta gdyż może się inaczej niż on zachowywać.

Podsumowując, zasada podstawiania Liskov pomaga uniknąć błędów wynikających z odmiennego niż zamierzone działania programu. Dzięki przestrzeganiu tej zasady otrzymujemy: czytelniejszy i łatwiejszy w utrzymaniu kod oraz możliwość podstawienia dowolnej klasy pochodnej w miejsce klasy bazowej bez obaw, że program źle zadziała.

Zadania

Zadanie 1

1. Stwórz klasę o nazwie `TQuadEq` reprezentującą trójmian kwadratowy. Klasa powinna zawierać:
    a. pola reprezentująca współczynniki trójmianu
    b. Konstruktor parametryczny z przypisanymi wartościami domyślnymi argumentów
    c. Gettery oraz settery dla wszystkich pól klasy

d. metodę `void GetRoots(double& root1, double& root2) const` zwracającą rzeczywiste pierwiastki trójmianu.

e. W funkcji `main()` napisz prosty program demonstrujący wykorzystanie klasy `TQuadEq`. Program wczytuje z klawiatury współczynniki trójmianu a następnie wyświetla na ekranie jego pierwiastki.

Zadanie 2

2. Create a class named `MyString` which publicly inherits from the `string`. The class should contain:
   a. a constructor that takes `const char*` argument and prints on the screen the string "*MyString constructor called*". Assign a default value to the constructor argument.
   b. a destructor which prints on the screen the string "*MyString destructor called*"
   c. overloaded output operator
3. Create a class named `Figure`. The class should consist of:
   a. private floating point fields (e. g. `m_x, m_y`) that represent the figure location on a plane and the field named `m_label` of `MyString` type,
   b. a constructor that takes two double arguments `x` and `y` and `label` argument of type `const char*` and prints on the screen the string "*Figure constructor called*". Assign default values to all constructor arguments.
   c. destructor which prints on the screen the string "*Figure destructor called*",
   d. getter methods for the private fields.
   e. `void print(void)` method that prints on the screen the string "*I'm a Figure*" as well as the location and the label of the `Figure` object.
4. Create a class named `Rectangle` that publicly inherits from the `Figure` class. The class should have the following fields:
   a. private floating point fields (e. g. `m_w, m_h`) that represent rectangle width and height,
   b. a constructor that takes four double arguments `x`, `y`, `w`, `h` and `label` of type `const char*` and prints on the screen the string "*Rectangle constructor called*". Assign default values to all constructor arguments.
   c. a destructor that prints on the screen the string "*Rectangle destructor called*",
   d. getter methods for the private fields.
5. In the `main` function do:
   a. Create a `Figure` object using the default constructor, run the program and notice the order each constructor and destructor is being called.
   b. Create a `Rectangle` object using the default constructor, run the program and notice the order each constructor and destructor is being called.
   c. Create `Rectangle` object with any arguments (e. g. x=1.0, y=2.0, w=10.0 and label "*rectangle 1*")
   d. Call the `print` method from the `Rectangle` object. Notice the function being called.
   e. Add the `void print(void)` method to the `Rectangle` class that prints on the screen the string "*I'm a Rectangle*" as well as the location and the label of the `Rectangle` object and run the program. Notice the function being called.
   f. Add the `MyString& to_upper()` method to the `MyString` class which converts all characters in the `MyString` object to uppercase. Call the method in all above `print` functions.
6. Suppose we want to create a class named `Circle`. Which class (`Figure`, `Rectangle`) should `Circle` inherit from?