Programowanie obiektowe

Ćwiczenie nr. 4

Tworzenie klas. Implementacja metod klas.

dr inż. Adam Wolniakowski

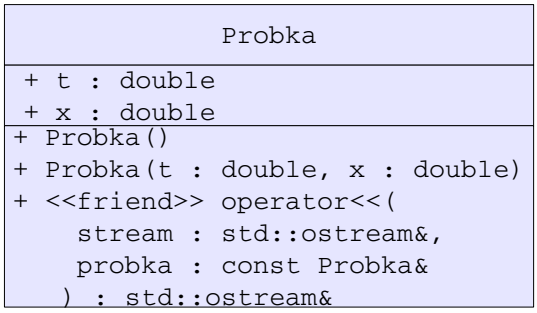
1. Cel ćwiczenia

Celem ćwiczenia jest zapoznanie się z podstawami programowania obiektowego: tworzeniem klas i implementacją metod klas.

2. Informacje

2.1. Diagram klasy

Diagram klas w języku UML opisuje statyczną strukturę projektu, w szczególności występujące w projekcie klasy, ich składowe i zależności. Pojedyncza klasa jest reprezentowana przez prostokąt:



W górnej części prostokąta znajduje się nazwa klasy. W części poniżej wypisane są pola klasy (dane). W najniższym prostokącie opisane są metody klasy. Dostępność składowych oznaczana jest przez znaki: + (publiczna), # (chroniona) i – (prywatna).

Pola opisane są w następujący sposób:

```
+ [nazwa pola] : [typ pola]
```

Na przykład zapis "+ t : double" oznacza, że definiujemy publiczne pole klasy: "double t;".

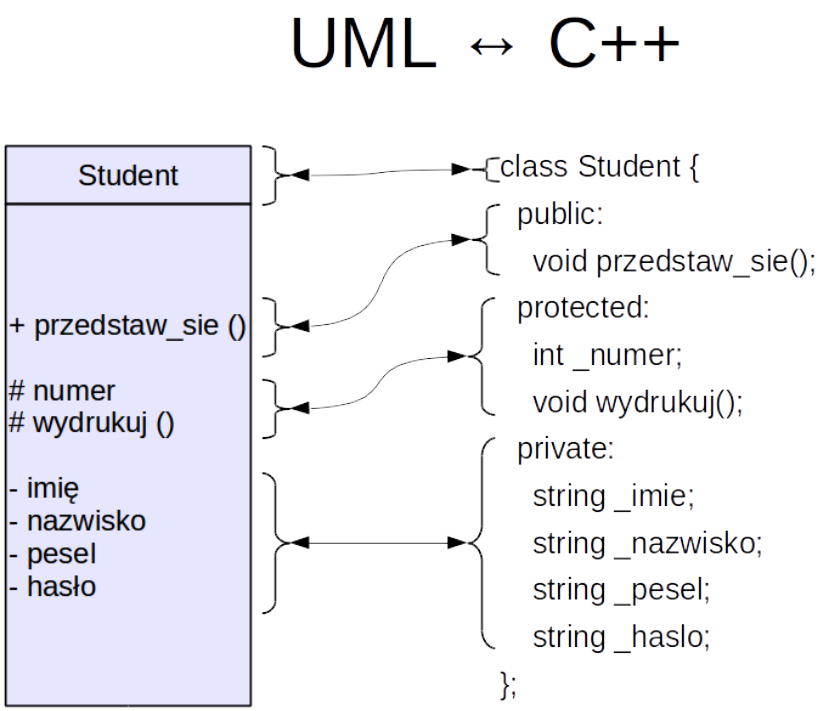
Metody klasy opisane są następująco:

```
+ [<<specyfikator>>] [nazwa metody] ([argumenty...]) : [typ zawracanej warto?ci]
```

Na przykład zapis "+ srednia (sygnal : const Sygnal&) : double" oznacza, że definiujemy metodę publiczną o następującej sygnaturze: "double srednia(const Sygnal& sygnal);"

Jeżeli metoda nazywa się tak samo jak klasa i nie zaznaczono jaki jest typ zwracanej wartości, jest to prawdopodobnie konstruktor klasy. Specyfikator <<friend>> oznacza, że mamy do czynienia z tzw. *funkcją zaprzyjaźnioną* (patrz: 2.4).

Diagram klasy przekształcamy w plik nagłówkowy w następujący sposób:



2.2. Powiązania w diagramie klas

Strzałki na diagramie klas oznaczają powiązania między klasami. Na razie nie będziemy się nimi specjalnie interesować. Warto pamiętać, że jeżeli na diagramie zaznaczono strzałkę, onacza to, że prawdopodobnie w pliku nagłówkowym zależnej klasy należy załączyć (#include) plik nagłówkowy klasy wskazywanej strzałką.

Przykładowe typy powiązań:

- "używa" – klasa używa innej klasy w jakimś celu. Strzałka może posiadać opis precyzyjniej formułujący zależność, np. <<create>> – klasa tworzy obiekty klasy wskazywanej strzałką, <<use>> f – klasa operuje na tej wskazywanej strzałką, ⇢
- "składa się z" – klasa zawiera klasę wskazywaną strzałką jako integralną składową (np. Sygnał zawiera Próbki). Specyfikator krotności (np. 0..*) określa ile elementów może być składowymi. ◆—

2.3. Przeładowanie operatora []

Można zdefiniować zachowanie operatorów dla klas stworzonych przez użytkownika. Na przykład, jeśli wewnątrz klasy zdefiniujemy zachowanie operatora [] w następujący sposób:

```
class MojaTablica {
public:
  int& operator[](int indeks) {
    return _dane[indeks];
  }

private:
  std::vector<int> _dane;
};
```

możliwe będzie indeksowanie klasy MojaTablica tak jakby była tablicą:

```
MojaTablica tab;
cout << tab[0] << endl;
```

2.4. Przeładowanie operatora <<

Domyślnie, nie jest możliwe wypisanie instancji samodzielnie zdefiniowanej klasy przy pomocy operatora <<. Aby to zmienić, konieczne jest zdefiniowanie zachowania operatora << dla danej klasy:

```
class Student {
public:
   friend std::ostream& operator<<(std::ostream& strumien, const
Student& student);
private:
  std::string _imie;
  std::string _nazwisko;
};
std::ostream& operator<<(std::ostream& strumien, const Student&
student) {
     stream << "Student: " << student._imie << " " <<
student._nazwisko;
  return stream;
}
```

Teraz możliwe jest bezpośrednie wypisanie obiektu klasy Student np. na cout:

```
Student nowy;
cout << nowy << endl;
```

Uwaga: Specyfikator *friend* oznacza, że pomimo że funkcja nie jest metodą klasy, ma dostęp do jej prywatnych składowych. Funkcja taka **nie jest** metodą klasy, dlatego przy jej definicji nie podajemy specyfikatora zakresu.

3. Zadanie

Należy zmodyfikować projekt programu operującego na sygnałach utworzony na poprzednich zajęciach. Zaleca się pracę na tym samym repozytorium (np. poprzez utworzenie nowej gałęzi *branch* projektu). Dla projektu należy nadać strukturę obiektową, przedstawioną na załączonym diagramie UML. Dla każdej klasy należy utworzyć oddzielny plik nagłówkowy *.hpp oraz plik z implementacją *.cpp. W plikach nagłówkowych należy umieścić *include guardy*, deklaracje klas i funkcji oraz dokumentację systemu Doxygen. W plikach *.cpp należy zawrzeć definicje i implementacje metod klas. Należy sprawdzić i zaprezentować działanie programu a następnie wygenerować dokumentację.