# Języki i metody programowania 2

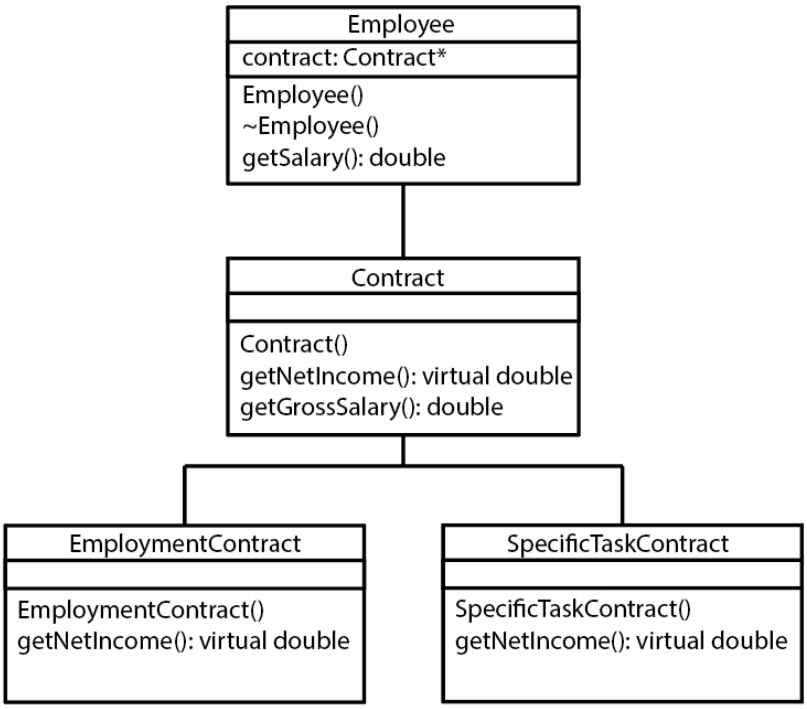
Lab 9

## Polimorfizm

Polimorfizm to możliwość różnej odpowiedzi na ten sam komunikat (wywołanie tej samej metody) przez obiekty różnych klas powiązanych dziedziczeniem. Poniższe podsekcje składają się na opis tego mechanizmu.

## Funkcje wirtualne

Podczas dziedziczenia, klasy pochodne mogą nadpisywać metody swoich klas bazowych. Kiedy jednak dokonujemy rzutowania w górę (np. z LiczbaUrojona na LiczbaRzeczywista), nadpisane metody wracają z powrotem do swoich pierwotnych postaci (np. obliczanie pola przekroju powierzchni kuli za pomocą rzutowania na koło). Takie zachowanie nie zawsze jest pożądane.
Załóżmy, że chcemy zbudować prosty program obsługujący bazę **pracowników** pewnej firmy. Zakładamy, że firma zatrudnia pracowników na **umowę o dzieło (SpecificTaskContract)**, lub na **umowę o pracę (EmploymentContract)**; wpływa to na obliczenie pensji netto każdego pracownika.
Mamy zatem taką relację:




Taka relacja umożliwia stworzenie tylko jednej klasy *Employee*, która będzie miała jedno pole *Contract*, któremu z kolei będzie przypisywana albo umowa o dzieło (*SpecificTaskContract)* albo umowa o pracę *(EmploymentContract)*. W takim wypadku po rzutowaniu *SpecificTaskContract* lub *EmploymentContract* na typ bazowy *Contract*, musi być możliwe wywołanie metod odpowiednio obliczających wynagrodzenie netto. W tym celu stosowane są **funkcje wirtualne**.
**Funkcja wirtualna**, to taka metoda klasy, która nadpisana w klasie pochodnej, nawet po rzutowaniu obiektu na typ bazowy, zachowa swoją implementacje. Klasa bazowa z funkcją wirtualną będzie *polimorficzna* - w zależności od tego jaki obiekt klasy pochodnej był na nią rzutowany, taka implementacja metody zostanie uruchomiona.

Przykład:

```
#include <iostream>
#include <list>
#include <memory>

using namespace std;

class Contract{
  public:
    Contract(double salary):grossSalary(salary){};
    virtual double getNetIncome() const;
    double getGrossSalary() const;
  protected:
    double grossSalary;
};

class SpecificTaskContract: public Contract{
  public:
    SpecificTaskContract(double salary):Contract{salary} {};
    virtual double getNetIncome() const override;
};

class EmploymentContract: public Contract {
  public:
    EmploymentContract(double salary):Contract{salary} {};
    virtual double getNetIncome() const override;
};

class Employee{
  public:
    Employee(){…};
    double getSalary() const;
  private:
    std::unique_ptr<Contract> contract_;
};
```

## Destruktory wirtualne

Zwróć uwagę, że podobnie do metod zachowują się destruktory. Jeśli obiekt jest jawnie niszczony przez operator *delete* to wywoływany jest jego destruktor:

```
#include <iostream>
(...)
int main(){
  Contract* contract = new EmploymentContract(10000);

  // zostanie wywołany destruktor klasy Contract, a nie EmploymentContract!
  // Jeśli EmploymentContract alokowałaby dodatkowo jakąś pamięć
  // NIE ZOSTAŁABY ONA ZWOLNIONA
  delete contract;
}
```

Rozwiązaniem jest deklarowanie destruktora jako wirtualnego:

```
class Contract{
  ...
  virtual ~Contract();
};

class EmploymentContract : public Umowa{
  ...
  virtual ~EmploymentContract();
```

```
};

int main(){
  Contract* contract = new EmploymentContract(10000);

  // zostanie wywołany destruktor klasy EmploymentContract
  delete contract;
}
```

## Klasy abstrakcyjne

Czasem w klasie bazowej implementowanie jakiejś metody jest bezcelowe. W naszym przykładzie z pracownikami implementowanie metody *getNetIncome* w klasie *Contract* nie ma sensu, bo nie wiadomo jaki jest to typ umowy.

Metody takie można zadeklarować jako czysto wirtualne:

```
class SpecificTaskContract: public Contract{
  protected:
    (…)
  public:
    SpecificTaskContract(double salary) {(…)};

    // Metoda abstrakcyjna. Nie posiada implementacji w klasie bazowej.
    // MUSI jednak być zaimplementowana w pochodnej lub ponownie
    // zadeklarowana jako abstrakcyjna.
    virtual double getNetIncome() = 0;
};
```

Klasę zawierającą choć jedną metodę abstrakcyjną nazywamy **klasą abstrakcyjną**. Nie jest możliwe utworzenie obiektu takiej klasy! Służy ona jedynie jako klasa bazowa dla innych klas. Swego rodzaju interfejs.

## Zadania

1. Zdefiniuj klasę abstrakcyjną **Query** z pojedynczą metodą abstrakcyjną

```
bool Accept(const Student &student);
```

Klasa Query reprezentuje ogólne zapytanie do repozytorium studentów. Klasa repozytorium powinna udostępniać nową metodę

```
std::vector<Student> FindByQuery(const Query &query)
```

która przegląda wszystkich zgromadzonych studentów i każdego po kolei przekazuje do zaakceptowania, jeśli student został zaakceptowany powinien znaleźć się w wynikowym wektorze. Zdefiniuj następnie implementacje zapytań:

```
ByFirstName
ByLastName
ByOneOfPrograms
ByYearLowerOrEqualTo

OrQuery
AndQuery
AndQuery(std::unique_ptr<Query> left, std::unique_ptr<Query> right)
```