Opracowanie: dr inż. Łukasz Maliński i mgr inż. Błażej Nycz, Politechnika Śląska, 2022

# LABORATORIUM 2.3
# INTERAKCJE OBIEKTOWE

**Poruszane zagadnienia z dziedziny programowania:**

- Podstawowe interakcje miedzy klasami - **wykład 7,**

**Umiejętności do opanowania:**

- implementacja wybranej relacji jednostronnej: asocjacja, lub agregacja
- implementacja kompozycji i dostępu do komponentów z poza klasy kompozytu.
- implementacja asocjacji obustronnej w warunkach podziału programu na jednostki translacji.

**Oznaczenia odnośnie samodzielności pracy:**

■ – **Ten problem koniecznie rozwiąż w pełni samodzielnie.** Możesz posługiwać się literaturą, wykładami i dokumentacją w celu sprawdzenia niuansów składniowych, ale nie szukaj gotowych rozwiązań i algorytmów. **Jest to bardzo ważne z punktu widzenia nauki i oszukiwanie przyniesie tylko poważne braki w późniejszych etapach nauki!**

▲ – Rozwiązując ten problem możesz posiłkować się rozwiązaniami zapożyczonymi od innych programistów, **ale koniecznie udokumentuj w kodzie źródło.** Nie kopiuj kodu bezmyślnie, tylko dostosuj go do kontekstu zadania. **Koniecznie musisz dokładnie zrozumieć rozwiązanie, które adoptujesz, gdyż inaczej wiele się nie nauczysz!**

● – Rozwianie tego problemu właśnie polega na znalezieniu i użyciu gotowego rozwiązania, **ale koniecznie podaj źródło.** Nie będzie wielkiej tragedii jeśli wykażesz się tylko ograniczonym zrozumieniem rozwiązania, gdyż nie musi być ono trywialne.

**Przykładowe zadania do rozwiązania w ramach samodzielnej nauki (nie oceniane):**

**Zadanie A (z rozwiązaniem):**

■ Napisz klasę **Przyrząd**, która będzie przechowywać dwa dowolne parametry strojenia (liczby rzeczywiste). W klasie umieść właściwe akcesory do pól.

*Wskazówki: Dwa pola tego samego typu można przechowywać w tablicy. To jest dobre rozwiązanie gdyż pozwala łatwo dodawać nowe parametry. W takim przypadku można też ograniczyć się do jednego setera i jednego getera, które będą przyjmować numer parametru który chcemy zapisać/odczytać.*

■ Następnie napisz klasę **Kalibrator**, która pozwoli na kalibrację dowolnej liczby **Przyrządów** (zebranych w tablicy), na te same wartości parametrów. Klasa powinna zawierać pola do przechowania zadanych wartości kalibrowanych parametrów, oraz metodę kalibruj, przejmującą tablicę klasy **Przyrząd**.

*Wskazówki: Wartości do kalibracji powinny być przechowane w polach klasy kalibrator, a nie przekazywane przy każdej kalibracji ręcznie. Klasa Kalibrator powinna używać klasy Przyrząd, ale tylko przelotnie – jedna metoda powinna korzystać z tej klasy.*

Przykładowe rozwiązanie:

```
#include<cmath>
#include <stdexcept>

using typParametru = double;
const unsigned int LICZBA_PARAMETROW = 2;

class Przyrzad
{
private:
  typParametru parameter[LICZBA_PARAMETROW];

public:
```

```
  typParametru getParameter(int nrParametru = 0) const
  {
    if (nrParametru < 0 || LICZBA_PARAMETROW < nrParametru)
      throw std::out_of_range("Niewlasciwy numer parametru");
    return parameter[nrParametru];
  }
  void setParameter(typParametru iParameter, int nrParametru)
  {
    if (0 <= nrParametru && nrParametru <= LICZBA_PARAMETROW)
      parameter[nrParametru] = iParameter;
  }
};

class Kalibrator
{
private:
  typParametru wartosci[LICZBA_PARAMETROW] = {};

public:
  void setParametry(typParametru* wart)
  {
    if (wart != nullptr)
      memcpy(wartosci, wart, sizeof(typParametru) * LICZBA_PARAMETROW);
  }



  void Kalibruj(Przyrzad* przyrzady, int liczba)
  {
    if (przyrzady != nullptr && liczba > 0)
      for (size_t przyrzad = 0; przyrzad < liczba; przyrzad++)
        for (size_t parametr = 0; parametr < LICZBA_PARAMETROW; parametr++)
          przyrzady[przyrzad].setParameter(wartosci[parametr], parametr);
  }
};

/* Zauważ, że mamy tu relację Zależności - klasa Kalibrator jest zależna od klasy Przyrząd.
   Kalibrator używa klasy Przyrząd w metodzie kalibruj. Koncepcja relacji jest taka, że
   ustawiamy w kalibratorze pożądane wartości parametrów Przyrządu. Następnie dokonując
   kalibracji, przesyłamy do niej tablicę przyrządów, na których działa metoda, która
   ustawia parametry we wszystkich wskazanych przyrządach na zadane wcześniej wartości. */
```

**Zadanie B (z rozwiązaniem):**

▲ Rozbuduj klasy **Autor** i **Dzieło** ze slajdu 15 (wykład 7) tak, aby każdy **Autor** mógł mieć przypisane do 10 różnych **Dzieł**, a każde dzieło do 5 **Autorów**.

*Wskazówki: Definiując asocjację stosuj zwykłe tablice (nie dynamiczne), Dzieła u Autora przypisuj na pierwszej wolnej pozycji w tablicy. W przypadku Autorów, daj opcję wskazania, którego z pięciu Autorów aktualnie podajesz. W razie nie podania numeru Autora przypisz go na pierwszej wolnej pozycji.*

▲ Dodaj do klasy metodę do rozdzielania **Autora** i **Dzieła** przez porównanie tożsamościowe.

*Wskazówki: Pamiętaj, że każde przypisanie Dzieła do Autora powinno też przypisać Autora do Dzieła i odwrotnie. Jeśli to niemożliwe zgłoś wyjątek.*

Przykładowe rozwiązanie:

```
const size_t MAX_LICZ_DZIEL = 10, MAX_LICZ_AUTOROW = 5;

class Dzielo; class Autor;

class Autor
{
private:
  Dzielo* Dziela[MAX_LICZ_DZIEL] = {};
```

```
public:
  const Dzielo* getDzielo(int iKtore) const
  {
    if (0 <= iKtore && iKtore < MAX_LICZ_DZIEL )
      return Dziela[iKtore];
    else throw std::out_of_range("Niewlasciwy numer dzieła");
  }
  void dodajDzielo(Dzielo* iDzielo);
  void usunDzielo(Dzielo* iDzielo);
};

class Dzielo
{
private:
  Autor* autorzy[MAX_LICZ_AUTOROW] = {};

public:
  const Autor* getAutor(int iKtory) const
  {
    if (0 <= iKtory && iKtory < MAX_LICZ_AUTOROW)
      return autorzy[iKtory];
    else throw std::out_of_range("Niewlasciwy numer dzieła");
  }
  void dodajAutora(Autor* iAutor, int iKtory = -1);
  void usunAutora(Autor* iAutor);
};

void Autor::dodajDzielo(Dzielo* iDzielo)
{
  /* Sprawdź czy Dzieło już jest przypisane: */
  for (size_t nrDziela = 0; nrDziela < MAX_LICZ_DZIEL; nrDziela++)
    if (Dziela[nrDziela] == iDzielo) return;

/* Jeśli Dzieło nie jest już przypisane to znajdź wolną pozycję: */
  for (size_t nrDziela = 0; nrDziela < MAX_LICZ_DZIEL; nrDziela++)
  {
    if (Dziela[nrDziela] == nullptr)
    {
       /* Przypisz Dzieło do Autora: */
       Dziela[nrDziela] = iDzielo;
       /* Przypisz Autora do Dzieła:*/
       iDzielo->dodajAutora(this);
       return;
     }
   }
   throw std::length_error("Autor nie moze miec wiecej dziel");
}

void Autor::usunDzielo(Dzielo* iDzielo)
{
  /* Przeszukaj wszystkie Dzieła Autora:*/
  for (size_t nrDziela = 0; nrDziela < MAX_LICZ_DZIEL; nrDziela++)
  {
    /* Jeśli znajdziesz zadane Dzieło (porównanie tożsamościowe): */
    if (Dziela[nrDziela] == iDzielo)
    {
      /* Usuń przypisanie Dzieła do Autora:*/
      Dziela[nrDziela] = nullptr;

      /* Usuń przypisanie Autora do Dzieła: */
      iDzielo->usunAutora(this);
      return;
    }
  }
}
```

```
void Dzielo::dodajAutora(Autor* iAutor, int iKtory)
{
  /*Sprawdź czy taki Autor nie jest już przypisany: */
  for (size_t nrAutora = 0; nrAutora < MAX_LICZ_AUTOROW; nrAutora++)
    if (autorzy[nrAutora] == iAutor) return;

  /* Jeśli wskazano poprawny numer Autora:*/
  if (0 <= iKtory && iKtory < MAX_LICZ_AUTOROW)
  {
    /* Jeśli na tej pozycji nie ma już przypisanego innego Autora: */
    if (autorzy[iKtory] == nullptr)
    {
       /* Przypisz Autora do Dzieła: */
       autorzy[iKtory] = iAutor;

       /* Przypisz Dzieło do Autora: */
       iAutor->dodajDzielo(this);
       return;
    }
    else throw std::invalid_argument("Proba nadpisania autora");
  }
  else /* Wyszukaj pierwszą wolną pozycję: */
  {
    for (size_t nrAutora = 0; nrAutora < MAX_LICZ_AUTOROW; nrAutora++)
    {
       /* Jeśli znajdziesz wolną pozycję: */
       if (autorzy[nrAutora] == nullptr)
       {
          /* Przypisz autora do Dzieła: */
          autorzy[nrAutora] = iAutor;

          /* Przypisz dzieło do Autora: */
         iAutor->dodajDzielo(this);
         return;
       }
    }
    throw std::length_error("Dzielo nie moze miec wiecej autorow:");
  }
}

void Dzielo::usunAutora(Autor* iAutor)
{
  /* Przeszukaj wszystkich Autorów Dzieła:*/
  for (size_t nrAutora = 0; nrAutora < MAX_LICZ_AUTOROW; nrAutora++)
  {
    /* Jeśli znajdziesz zadanego Autora (porównanie tożsamościowe): */
    if (autorzy[nrAutora] == iAutor)
    {
      /* Usuń przypisanie Autora do Dzieła:*/
      autorzy[nrAutora] = nullptr;

      /* Usuń przypisanie Dzieła do Autora:*/
      iAutor->usunDzielo(this);
      return;
    }
  }
}
```

**Zadanie C (z rozwiązaniem):**

▲ Zdefiniuj klasę **Punkt**, wzorując się na przekładzie ze saldu 21 (wykład 7). Klasa powinna przechowywać współrzędne w dwuwymiarowym układzie kartezjańskim. Wyklucz w tej klasie konstruktor domyślny!

*Wskazówki: Wykluczenie konstruktora domyślnego najlepiej zrobić jawnie. Współrzędne przechowuj w osobnych polach i koniecznie utwórz do nich akcesory.*

■ Zdefiniuj klasę **Linia**, będącą kompozytem dwóch **Punktów**: początku i końca. Instancja klasy **Linia** powinna być tworzona przez zadanie dwóch instancji punktów, lub poszczególnych współrzędnych (4 liczby rzeczywiste). Wyklucz w tej klasie konstruktor domyślny!

*Wskazówki: Definiując konstruktory dla klasy Linia koniecznie wywołuj konstruktory klasy Punkt, na liście inicjalizacyjnej.*

■ Zdefiniuj w klasie **Linia** metodę zwracającą jej długość tak, aby dało się ją wywołać dla instancji chronionej przed zapisem.

*Wskazówki: Koniecznie użyj akcesorów do współrzędnych w klasie Punkt. Zauważ, że ochrona przed zapisem intonacji Linia, propaguje na jej pola. Oznacza to, że getery współrzędnych w klasie Punkt, także muszą być dostosowane do pracy z instancją chronioną przed zapisem.*

Przykładowe rozwiązanie:

```
class Punkt
{
  float m_x, m_y;
public:
  Punkt() = delete;
  explicit Punkt(float x, float y)
  {
    setXY(x, y);
  }
  void setXY(float x, float y)
  {
    m_x = x;
    m_y = y;
  }
  int getX(void) const { return m_x; }
  int getY(void) const { return m_y; }
};

class Linia
{
  Punkt poczatek;
  Punkt koniec;

public:
  Linia() = delete;

  Linia(Punkt iPoczatek, Punkt iKoniec)
    : poczatek(iPoczatek), koniec(iKoniec)
  {}

  Linia(float iXPoczatek, float iYPoczatek, float iXKoniec, float iYKoniec)
    : Linia(Punkt(iXPoczatek, iYPoczatek), Punkt(iXKoniec, iYKoniec))
  {}

  float dlugosc() const
  {
     return sqrt(pow(poczatek.getX() - koniec.getX(), 2) +
                 pow(poczatek.getY() - koniec.getY(), 2));
  }
};

int main(void)
{
  const Linia instancjaStala(0.0, 0.0, 2.0, 3.0);
  cout << instancjaStala.dlugosc() << endl;


}
```

**Zadania domowe (oceniane)**

**UWAGA: Można oddać tylko jedno zadanie domowe. Ocena za zadanie domowe jest uwzględniana tylko jeśli uzyska się przynajmniej 1,0 pkt za rozwiązanie zadań podanych na zajęciach!**

**W żadnym z zadań nie można używać std::string!**

**Zadanie 1 [1,0 pkt]:**

■ Zdefiniuj klasę **PESEL**, której zadaniem będzie przechowywanie numeru pesel dla osoby. Klasa powinna dostarczać możliwość zadania numeru pesel tylko w chwili tworzenia instancji (nie twórz setera) i powinna weryfikować, czy podany numer jest prawidłowy. W klasie nie absolutnie powinno być konstruktora domyślnego!

*Wskazówki: Zauważ że numer pesel składa się zawsze z 11 cyfr i każda cyfra jest ważna. Numer także może zaczynać się od 0 i to zero jest istotne. Mając to na uwadze zastanów się jak właściwie dobrać typ pola. Poczytaj o tym jak konstruuje się numer pesel, oraz jak sprawdza się jego poprawność.*

■ Dla klasy **PESEL** zdefiniuj, metody pozwalające na odczytanie:

- ile lat ma właściciel numeru, w chwili wywołania tej metody,
- jakiej płci jest właściciel numeru.

*Wskazówki: Zauważ że, odczytanie wieku wymaga poznania aktualnej daty – poszukaj jak odczytać aktualną datę z zegara systemowego.*

■ Zdefiniuj prostą klasę **Osoba**, przechowującą imię, nazwisko i instancję klasy **PESEL**. Wszystkie dane osoby powinny móc być zadawane tylko przy jej tworzeniu i numer pesel nie może być zadawany jako instancja klasy PESEL. W klasie nie absolutnie powinno być konstruktora domyślnego!

*Wskazówki: Koniecznie skorzystaj tu z relacji kompozycji. Nie definiuj seterów dla tej klasy, ale getery tak. Zastanów się jak przesyłać numer pesel do konstruktora tak, aby nie trzeba było wcześniej tworzyć jego oddzielnej instancji.*

■ Dla klasy **Osoba** zdefiniuj operator << określający jak cout ma wyświetlić dane osoby. Format wyświetlania danych dobierz według uznania, ale powinny wyświetlić się informacje takie jak: imię, nazwisko, aktualny wiek i płeć.

*Wskazówki: Zwróć uwagę, że właściwe funkcjonalności są już dostarczane przez klasę PESEL – trzeba ich tylko umiejętnie użyć.*

**Ocenianie:**

| | |
|---|---|
| Projekt klasy PESEL: | 0,2 pkt. |
| Odczyt danych z nr pesel: | 0,2 pkt. |
| Projekt klasy Osoba (kompozycja): | 0,2 pkt. |
| Operator <<: | 0,2 pkt. |
| Ogólna jakość kodu: | 0,2 pkt. |

**Zadanie 2 [2,0 pkt]:**

■ Samodzielnie zaprojektuj klasę **Sejf**, która przechowywać będzie tablicę danych dowolnego typu (wybierz jakiś konkretny typ i rozmiar tablicy). Klasa powinna mieć także komponent typu **ZamekCyfrowy**, który odpowiedzialny będzie za przechowywanie i weryfikację klucza dostępu (dowolna informacja która zajmuje nie mniej niż 4 bajty w pamięci), oraz wskazanie do 3 **Osób** (pusta klasa) uprawnionych do otwierania **Sejfu**.

Projekt klas powinien spełniać następujące wytyczne:

- Przy próbie odczytu/zapisu danych zawsze trzeba podać wskazanie **Osoby**, która próbuje tej operacji dokonać, oraz cyfrowy klucz dostępu.
- Cyfrowy klucz dostępu jest nadawany fabrycznie - powinien być ustalany raz w chwili tworzenia instancji **ZamkaCyfrowego**. Klucza nie można już zamienić

w trakcie istnienia instancji i absolutnie nie powinno się dać go odczytać poza klasą.

- Instancja klasy **ZamekCyfrowy** powinna komponentem klasy **Sejf** i być wypełni odpowiedzialna za przechowywanie i weryfikację poprawności klucza. Klasa ta musi zgłosić do klasy **Sejf** zgodę na dostęp do danych. **ZamekCyfrowy** nie może udostępnić klucza nawet klasie **Sejf**. To klasa **Sejf** przesyła mu podany przez użytkownika klucz do weryfikacji wewnętrznej.
- Jeśli zdefiniowano choć jedną osobę upoważnioną to, podanie samego klucza nie wystarczy do uzyskania dostępu do danych. Należy jeszcze wskazać instancję klasy **Osoba**, która chce uzyskać dostęp. **Osoba** musi znajdować się na „liscie" (słowo lista nie oznacza tu typu struktury danych), osób upoważnionych.
- **Osoby** upoważnione można swobodnie zmieniać w trakcie istnienia instancji klasy **Sejf**.

*Wskazówki: Relacja Sejf-ZamekCyforwy to kompozycja, relacja Sejf-Osoba to asocjacja. Klasa Osoba nie musi być jakoś szczególnie projektowana, gdyż zależy nam tylko na porównaniu tożsamościowym (najlepiej gdy będzie to pusta klasa). Zarówno Sejf jak i ZamekCyfrowy zawsze wymagają utworzenia z podaniem jakiś danych (np. klucza dostępu), dlatego w tych klasach nie powinno być konstruktora domyślnego. Tablica danych powinna być zwykłą tablicą o stałym rozmiarze. Format klucza dostępu i sposób jego weryfikacji jest absolutnie dowolny.*

■ Zademonstruj w programie głównym działanie klasy Sejf.

*Wskazówki: Pokaż działanie Sejfu, gdy są przypisane osoby upoważnione i gdy nie.*

**Ocenianie:**

| | |
|---|---|
| Projekt klasy ZamekCyfrowy: | 0,6 pkt. |
| Kompozycja Sejf-ZamekCyfrowy: | 0,5 pkt. |
| Asocjacja z klasą Osoba: | 0,5 pkt. |
| Prezentacja w programie głównym: | 0,2 pkt. |
| Ogólna jakość kodu: | 0,2 pkt. |