# Zestaw 5

**1.** Zadanie testowe. Mamy kod (fragment):

```
#include <iostream>
using namespace std;
#define P(x) std::cout << x << std::endl

class A1 {
    int i { 5 };
public:
    A1(bool) { P("A1 c-tor"); }
    virtual ~A1() { P("A1 d-tor"); }
    int get() const { return i; }
};

class V1 : virtual public A1 { public:
    V1(bool) : A1(true) { P("V1 c-tor"); }
    virtual ~V1() override { P("V1 d-tor"); }
};

class C1 : virtual public V1;
class B2;
class B3;
class V2 : virtual public A1, public B2;
class C2 : virtual public V2, public B3;
class M1;
class M2;
class X : public C1, public C2 {
  M1 m1;
  M2 m2;
  public:
      // zaimplementuj c-tor i d-tor
};

int main() {
    A1 *pa = new X;
    cout << pa->get() << endl; // zakomentować gdy public A1
    delete pa;
}
```

Proszę dopisać definicje klas (jak wyżej), podobnie jak w klasie V1 (czyli c-tor i d-tor). Wszystkie konstruktory mają mieć argument (bool), dzięki temu będzie wymagane ich wywołanie w odpowiednich miejscach (może być zawsze true). W zasadzie większa część pracy to kopiowanie i edytowanie. Proszę przestudiować kolejność wywoływania konstruktorów i destruktorów. Proszę sprawdzić warianty a) i b) pisząc w komentarzu kodu wynik sprawdzenia:

a) co się stanie jeśli `virtual ~V1(); (i inne, bez override) i ~A1(); (niewirtualne).`
b) co się stanie jeśli będzie dziedziczenie zwykłe `public A1`

Po zbadaniu powyższych przypadków, kod przesłać w wersji z obecnymi słowami kluczowymi (override, virtual – jak w ramkach powyżej).

**2.** W implementacji wzorca Obserwator (patrz kod z zajęć) użyte zostały zwykłe wskaźniki, np.

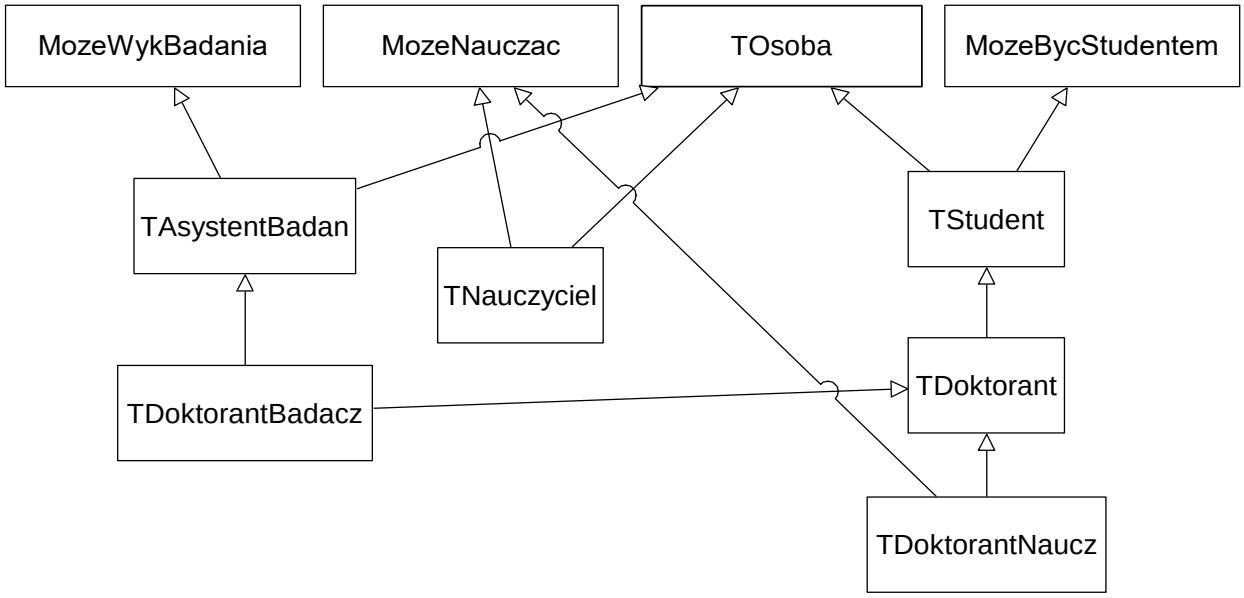
```
Obserwator* p1 = new Monitor("Monitor 1",stacja); // tak samo p2, p3, p4
```

Proszę przepisać kod w ten sposób, aby używany był unique_ptr:

```
unique_ptr<Obserwator> p1(new Monitor("Monitor 1",stacja));
```

Oczywiście trzeba odpowiednio dostosować wszystkie metody, które używają Obserwator*.
Po takiej zmianie zbędne będzie również `delete p1;` itd.

**3.** Napisz zestaw klas jak na diagramie, czyli „klasy mieszane" `MozeWykBadania`, `MozeNauczac`, `TOsoba`, `MozeBycStudentem` – oraz potomne, jak na diagramie (TNauczyciel opuścić). W każdej z klas umieść choć jedną metodę składową, która może jakoś odzwierciedlać cel czy działanie obiektu danej klasy. Zademonstruj działanie w programie, tworząc po kilka obiektów klas potomnych. Również przeanalizuj sytuację typu `TDoktorantBadacz`, który to typ otrzymuje klasę `TOsoba` na dwóch drogach dziedziczenia.




**4.** Proszę przestudiować slajdy 11-17 (uwaga: w pliku PDF wzorce projektowe) opisujące ewolucję wzorca Most (Bridge). Proszę zaprojektować i zademonstrować w programie ten wzorzec, kierując się diagramem ze slajdu 15. **Uwaga**: przykładowy kod programu, który powinien się wykonywać:

```
int main() {
    DP1 library1; // pierwsza biblioteka
    DP2 library2; // druga biblioteka

    Drawing *d1 = new V1Drawing( library1 );
    Drawing *d2 = new V2Drawing( library2 );

    Shape *p1 = new Rectangle( d1 );
    Shape *p2 = new Circle( d2 );

    p1->draw(); // rectangle linia V1
    p2->draw(); // circle okrag V2
    p1->setLib( d2 );
    p2->setLib( d1 );
    p1->draw(); // rectangle linia V2
```

```
    p2->draw(); // circle okrag V1

    delete p1;
    delete p2;
    delete d1;
    delete d2;
}
```

5. Proszę przestudiować slajdy 23-25 (uwaga: w pliku PDF wzorce projektowe) opisujące ewolucję wzorca Fabryka Abstrakcyjna (Abstract Factory). Proszę zaprojektować i zademonstrować w programie ten wzorzec, kierując się diagramem ze slajdu 25. Należy użyć obiekty unique_ptr jako inteligentne wskaźniki. **Uwaga**: przykładowy kod programu, który powinien się wykonywać:

```
int main() {
    unique_ptr<ResFactory> fabryka(new LowResFactory);
    unique_ptr<DisplayDrv> ddrv = fabryka->getDispDrv();
    unique_ptr<PrinterDrv> pdrv = fabryka->getPrintDrv();
    ddrv->wykonaj();
    pdrv->wykonaj();

    fabryka.reset(new HighResFactory);
    ddrv = fabryka->getDispDrv();
    pdrv = fabryka->getPrintDrv();
    ddrv->wykonaj();
    pdrv->wykonaj();
}
```