3.9. Programowanie zorientowane obiektowo

3.9.1. Klasy i obiekty

Definicja

Programowanie obiektowe jest techniką programowania, w której programy definiuje się za pomocą obiektów — elementów łączących stan (czyli dane, nazywane polami) i zachowanie (czyli metody). Obiektowy program komputerowy wyrażony jest jako zbiór takich obiektów, komunikujących się pomiędzy sobą w celu wykonywania zadań.

Klasa to typ obiektowy. **Definicja klasy** ma następującą postać:

```
class nazwa_klasy: tryb_dziedziczenia klasa_bazowa
{
 lista deklaracji pól
 lista deklaracji metod
};
```

gdzie:

`lista deklaracji pól` — zawiera deklaracje pól klasy będące deklaracjami danych lub struktur danych;

`lista deklaracji metod` — zawiera deklaracje metod klasy będące deklaracjami funkcji zawartych w klasie;

`tryb_dziedziczenia klasa_bazowa` — zawiera tryb dziedziczenia z klasy nadrzędnej w hierarchii klas.

Pola (czyli dane składowe) to zmienne zdefiniowane wewnątrz klasy, natomiast **metody** (czyli funkcje składowe) to funkcje zadeklarowane w klasie. Pola i metody zawarte w jednej klasie to **składowe klasy**. Dostęp do pól powinien odbywać się wyłącznie przy wykorzystaniu metod. Metody zawierają czynności wykonywane na polach.

Poznaliśmy już znaczenie pojęcia „klasa". Czym jest więc obiekt? **Obiekt** to złożona struktura danych o ustalonej liczbie elementów, na które składają się pola i metody. **Obiektem** nazywamy również zmienną typu obiektowego. Klasa jest typem obiektu, a nie obiektem. Definiując klasę, nie tworzymy żadnych obiektów.

Definicja

Klasę, której nie reprezentuje żaden obiekt, nazywamy **klasą abstrakcyjną**.

Deklaracje elementów składowych klasy można poprzedzić słowami kluczowymi określanymi jako **specyfikatory dostępu do składowych klasy**:

- `public`,
- `protected`,
- `private`.

Jeżeli składowe klasy nie są poprzedzone żadną z tych etykiet, kompilator przyjmuje **domyślnie specyfikator** `private`.

Specyfikatory sterują dostępem do składowych klasy. Składowe, które zadeklarowano w **sekcji** `public`, są dostępne w całym programie. Jeżeli umieścimy je w **sekcji** `private`, będziemy mieli do nich dostęp jedynie w funkcjach składowych klasy i w funkcjach zaprzyjaźnionych. Składowe zadeklarowane w **sekcji** `protected` są dostępne w funkcjach składowych klasy i w funkcjach składowych jej klas pochodnych (czyli dziedziczących).

Definicja

Hermetyzacja (inaczej enkapsulacja) to cecha programowania obiektowego polegająca na ukrywaniu wybranych składowych klasy przed bezpośrednim dostępem z zewnątrz. Do metod i pól ukrytych o ograniczonym dostępie (czyli znajdujących się w sekcjach `private` lub `protected`) możemy dotrzeć jedynie poprzez metody publiczne.

Przykład 3.58.

Napiszmy program (*prog3_45.cpp*), w którym zostanie zdefiniowana klasa `osoba` (zawierająca pola: `nazwisko`, `imie`, `wiek`, metody: `wczytaj()`, `wypisz()`) oraz utworzone zostaną dwa obiekty klasy `osoba`: `O1` i `O2`. Program ma realizować wczytywanie danych do pól i wypisywanie zawartości pól na ekranie. Dostęp do pól powinien być ograniczony — tylko poprzez metody. Pola należy więc umieścić w sekcji prywatnej, natomiast metody w sekcji publicznej.

```
#include <iostream>
#include <cstring>
using namespace std;
class osoba    //definicja klasy
{
  char nazwisko[25];    //sekcja prywatna
  char imie[20];
  int wiek;
```

```
 public:    //sekcja publiczna
  void wczytaj(char *n, char *i, int w);    //deklaracja metody
  void wypisz()    //definicja metody wewnątrz klasy
  {
   cout<<"\n"<<imie<<" "<<nazwisko<<endl;
   cout<<"wiek: "<<wiek<<endl;
  }
};    //koniec definicji klasy
void osoba::wczytaj (char *n, char *i, int w)    //definicja metody poza klasą
{
 strcpy(nazwisko,n);
 strcpy(imie,i);
 wiek=w;
}
int main()
{
 osoba O1, O2;
 char na[25], im[20];
 int wi;
 cout<<"podaj dane osoby nr 1:"<<endl;
 cout<<"nazwisko: ";
 cin.getline(na,sizeof(na));
 cout<<"imię: ";
 cin.getline(im,sizeof(im));
 cout<<"wiek: ";
 cin>>wi;
 O1.wczytaj(na,im,wi);
 O2.wczytaj("Kowalski","Jan",24);
 cout<<"\nwyświetlenie danych wszystkich osób:"<<endl;
 O1.wypisz();
 O2.wypisz();
 return 0;
}
```

Zauważ, że w zdefiniowanej klasie `osoba` pola są w sekcji **`private`**, natomiast metody są publiczne. Dostęp do pól mają więc tylko funkcje składowe tej klasy i właśnie nimi należy się posługiwać, aby korzystać z pól.

Metoda `wypisz()` została zdefiniowana wewnątrz definicji klasy. Funkcja ta wyświetla na ekranie wartości wszystkich pól. W praktyce wygodniej jest stworzyć dla każdego pola oddzielną metodę zwracającą jego wartość.

Funkcję składową `wczytaj()` zadeklarowano w klasie, natomiast jej definicja została umieszczona poza klasą. Spowodowało to konieczność zastosowania **operatora zasięgu** `::`, który wykorzystywany jest do podania nazwy klasy. W praktyce częściej stosuje się definiowanie metod poza klasą, ponieważ zapis jest czytelniejszy. Działanie metody `wczytaj()` polega na przypisaniu polom nowych wartości podanych w tej funkcji jako parametry. Tutaj również bardziej przydatne okazuje się przydzielenie każdemu polu oddzielnej metody ustawiającej jego wartość.

W programie utworzono dwa obiekty: `O1` i `O2`. Wartości pól obiektu `O1` wprowadzane są z klawiatury, natomiast obiektu `O2` — bezpośrednio w programie.

3.9.2. Konstruktory i destruktory

Definicja

Konstruktor jest specjalną metodą, uruchamianą automatycznie przy tworzeniu obiektu danej klasy. Zadaniem konstruktora jest zainicjowanie pól obiektu danej klasy, czyli przypisanie im wartości początkowych, przydzielenie pamięci czy wykonanie innych czynności niezbędnych do prawidłowego utworzenia obiektu. Nazwa konstruktora musi być taka sama jak nazwa zawierającej go klasy. Funkcja konstruktora nie ma określonego zwracanego typu, nie można więc przypisywać jej żadnej wartości.

Konstruktor nie jest obowiązkowym elementem definicji klasy. Jeżeli nie zostanie on zdefiniowany, kompilator automatycznie wygeneruje **konstruktor domyślny**. W praktyce jednak każda klasa powinna zawierać funkcję konstruktora.

W danej klasie można zdefiniować **więcej niż jeden konstruktor**, wykorzystując mechanizm przeładowania funkcji (patrz punkt 3.5.4, „Przeładowanie funkcji").

Przykład 3.59.

Poniżej przedstawiono definicję klasy analizowanej w przykładzie 3.58 z dodanymi deklaracjami dwóch konstruktorów (*prog3_46.cpp*).

```
class osoba
{
  char nazwisko[25];
  char imie[20];
  int wiek;
 public:
  osoba();    //deklaracja konstruktora 1.
  osoba(char *n, char *i, int w);    //deklaracja konstruktora 2.
  void wczytaj(char *n, char *i, int w);
  void wypisz();
};
```

Definicje tych konstruktorów mogą być następujące:

```
osoba::osoba()     //definicja konstruktora 1.
{
 strcpy(nazwisko,"");
 strcpy(imie,"");
 wiek=0;
}
osoba::osoba(char *n, char *i, int w)     //definicja konstruktora 2.
{
 strcpy(nazwisko,n);
 strcpy(imie,i);
 wiek=w;
}
```

Różnią się one parametrami (to właśnie one decydują o wyborze właściwego konstruktora przy tworzeniu obiektu klasy):

```
osoba O1;     //uruchomienie konstruktora 1.
osoba O2("Kowalski","Jan",24);     //uruchomienie konstruktora 2.
```

Definicja

Destruktor jest specjalną funkcją składową wywoływaną w chwili likwidacji obiektu danej klasy. Metoda ta jest funkcją bezparametrową, niezwracającą żadnej wartości. Nazwa destruktora składa się z nazwy klasy poprzedzonej znakiem ~. Do zadań destruktora należy zwalnianie zasobów wykorzystywanych przez obiekt i inne czynności porządkowe. Funkcja ta jest uruchamiana automatycznie przy usuwaniu obiektu danej klasy.

Destruktor **nie jest obowiązkowym** elementem klasy.

Nie wolno przeciążać funkcji destruktora, można go **zdefiniować tylko raz**.

Przykład 3.60.

Przeanalizujmy fragment programu (*prog3_47.cpp*) podany poniżej. Przedstawiony przykład klasy pokazuje zastosowanie destruktora. Wykorzystano tutaj definicję klasy z przykładów 3.58 i 3.59. Aby działanie destruktora było zauważalne, wprowadzono dodatkowo dynamiczny przydział pamięci dla tworzonych obiektów.

```
class osoba
{
  char nazwisko[25];
  char imie[20];
  int wiek;
```

```
 public:
  osoba(char *n, char *i, int w);
  ~osoba();    //deklaracja destruktora
  void wypisz();
};
```

Definicja destruktora w najprostszej postaci wygląda następująco:

```
osoba::~osoba()    //definicja destruktora
{}
```

W programie konstruktor uruchamiany jest wraz z operatorem **new**, natomiast destruktor przy wykonywaniu operacji **delete**. W przypadku obiektów dynamicznych odwołanie do składników klasy wymaga operatora **->**.

```
int main()
{
 osoba *wsk;
 wsk = new osoba("Kowalski","Jan",24);    //uruchomienie konstruktora
 wsk->wypisz();
 delete wsk;    //uruchomienie destruktora
 return 0;
}
```

Zadanie 3.48. Napisz program, w którym zostanie zdefiniowana klasa `przedmiot` zawierająca następujące elementy:

- pola: `nazwa`, `typ`, `producent`, `rok_produkcji`;
- metody: `wczytaj()`, `wypisz()`, dwa konstruktory.

Utwórz dwa obiekty klasy `przedmiot`: `P1` i `P2`. Program powinien realizować wczytywanie danych do pól i wypisywanie ich wartości na ekranie. Zastosuj następujący dostęp do składowych klasy:

- pola prywatne,
- metody publiczne.

Zadanie 3.49. Napisz program, w którym zostanie zdefiniowana klasa `ksiazka` zawierająca następujące elementy:

- pola: `tytul`, `autor`, `wydawnictwo`, `rok_wydania`, `cena`;
- metody: `wczytajTytul()`, `wczytajAutora()`, `wczytajWydawnictwo()`, `wczytajRokWydania()`, `wczytajCene()`, `wypiszTytul()`, `wypiszAutora()`, `wypiszWydawnictwo()`, `wypiszRokWydania()`, `wypiszCene()`, dwa konstruktory, destruktor.

Utwórz dwa obiekty klasy `ksiazka`: `K1` i `K2`, dla których zastosuj dynamiczny przydział pamięci. Program powinien wczytywać dane do pól i wypisywać ich wartości na ekranie. Zastosuj następujący dostęp do składowych klasy:

- pola prywatne,
- metody publiczne.

3.9.3. Dziedziczenie i hierarchia klas

Definicja

Dziedziczenie polega na ustanowieniu związku pomiędzy dwoma klasami, z których jedna jest **klasą bazową**, a druga **klasą pochodną**, czyli dziedziczącą. Po zdefiniowaniu takiej relacji klasa pochodna jest rozszerzana o wskazane składowe odziedziczone z klasy bazowej (czyli pola i metody). **Obiekty klasy pochodnej** zawierają więc składowe specyficzne dla tej klasy (zawarte w definicji klasy pochodnej) oraz pola i metody odziedziczone z klasy bazowej (określone w definicji klasy bazowej).

Klasa pochodna może dziedziczyć z więcej niż jednej klasy bazowej. Związek klasy pochodnej z jedną klasą bazową nazywamy **dziedziczeniem pojedynczym**. Jeśli jednak klasa pochodna jest związana z wieloma klasami bazowymi, mamy do czynienia z **dziedziczeniem wielokrotnym**. Wówczas w obiekcie klasy pochodnej występują, poza polami i metodami tej klasy, składowe odziedziczone ze wszystkich klas bazowych. Mechanizm ten umożliwia wiązanie ze sobą niezależnych od siebie typów klas.

Dziedziczenie może odbywać się pośrednio lub bezpośrednio. **Dziedziczenie bezpośrednie** występuje, gdy przodek danego obiektu jest zdefiniowany jako **niezależny typ obiektowy** (czyli nie dziedziczy z żadnej klasy). Jeśli jednak przodek sam dziedziczy elementy pewnego typu obiektowego, mówimy o **dziedziczeniu pośrednim**, ponieważ jego potomek dziedziczy pośrednio także elementy tego typu. Cecha ta pozwala na tworzenie **hierarchii klas**, która określa relacje pomiędzy poszczególnymi klasami.

Tryb dziedziczenia wskazywany jest za pomocą słów kluczowych `public`, `protected` lub `private`. Przy **dziedziczeniu publicznym**, określanym za pomocą słowa kluczowego `public`:

- składowe dziedziczone z sekcji `public` klasy bazowej dołączane są do sekcji publicznej klasy pochodnej;
- składowe dziedziczone z sekcji `protected` klasy bazowej dołączane są do sekcji `protected` klasy pochodnej.

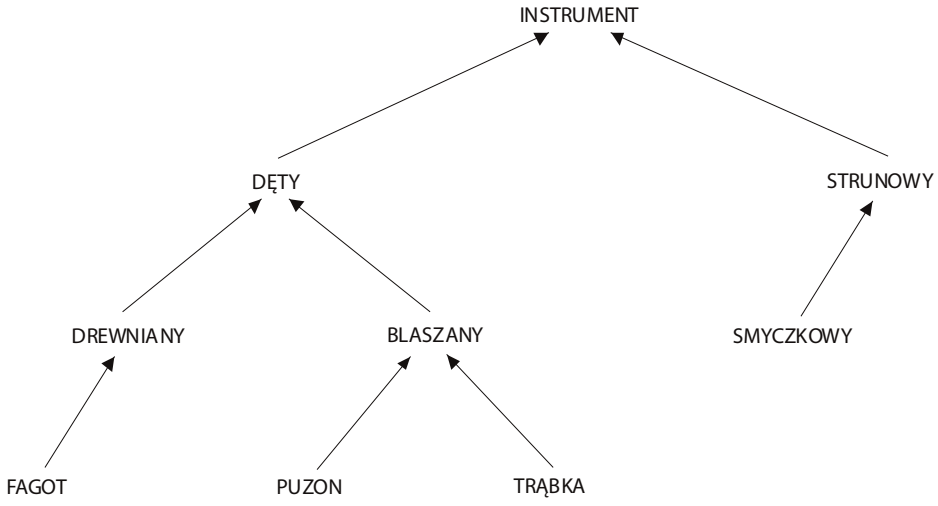
Przy **dziedziczeniu chronionym**, wskazywanym za pomocą słowa kluczowego `protected`, składowe dziedziczone zarówno z sekcji `public`, jak i z sekcji `protected` są dołączane do sekcji `protected` klasy pochodnej.

Przy **dziedziczeniu prywatnym**, określanym za pomocą słowa kluczowego `private`, składowe dziedziczone zarówno z sekcji `public`, jak i z sekcji `protected` klasy bazowej są dołączane do sekcji `private` klasy pochodnej.

Składowe sekcji `private` klasy bazowej w każdym wypadku są niedostępne dla klasy pochodnej.

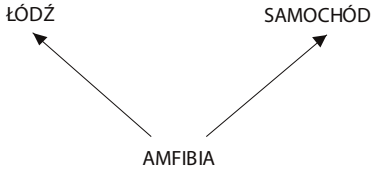
Przykład 3.61.

Przyjrzyj się przykładom hierarchii klas przedstawionym na rysunkach 3.19 i 3.20. Dokładnie przeanalizuj ich strukturę.



Rysunek 3.19. Przykładowa hierarchia klas z dziedziczeniem pojedynczym

W przedstawionej na rysunku 3.19 hierarchii klas występuje tylko dziedziczenie pojedyncze. Niezależną klasą bazową jest klasa `INSTRUMENT`. Występuje tutaj dziedziczenie bezpośrednie i pośrednie. Na przykład klasa `SMYCZKOWY` dziedziczy bezpośrednio z klasy `STRUNOWY` i pośrednio z klasy `INSTRUMENT`.



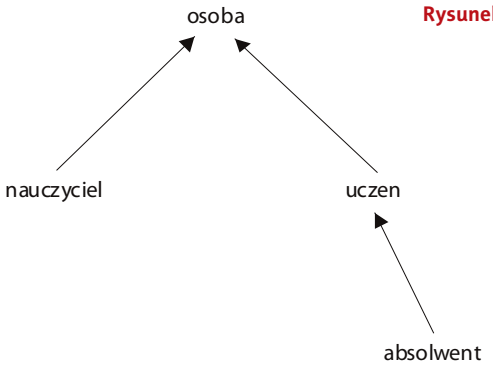
Rysunek 3.20. Przykładowa hierarchia klas z dziedziczeniem wielokrotnym

W hierarchii klas pokazanej na rysunku 3.20 występuje dziedziczenie wielokrotne. Klasa `AMFIBIA` dziedziczy bezpośrednio z dwóch klas: `ŁÓDŹ` i `SAMOCHÓD`. Amfibia, będąc pojazdem wodno-lądowym, może pełnić funkcję zarówno łodzi, jak i samochodu. Posiada więc cechy charakterystyczne dla obydwu wskazanych pojazdów.

Przykład 3.62.

Podane poniżej definicje klas tworzą hierarchię przedstawioną na rysunku 3.21.

Istnieje w tym przypadku możliwość tworzenia obiektów różnych klas. Mało prawdopodobne jest jednak, że zaistnieje potrzeba stworzenia obiektu klasy `osoba`. O klasie tej możemy więc powiedzieć, że jest **klasą abstrakcyjną**.



Rysunek 3.21. Hierarchia klas do przykładu 3.62

```
class osoba
{
  char nazwisko[30];
  char imie[20];
  int wiek;
 public:
  osoba(char *n, char *i, int w);
  ~osoba();
  char *wypiszNazwisko();
  char *wypiszImie();
  int wypiszWiek();
  void wczytajNazwisko(char *n);
  void wczytajImie(char *i);
  void wczytajWiek(int w);
};

class nauczyciel : public osoba
{
  int staz;
  char przedmiot[25];
 public:
  nauczyciel(char *n, char *i, int w, int s, char *p);
  ~nauczyciel();
  int wypiszStaz();
  char *wypiszPrzedmiot();
  void wczytajStaz(int s);
  void wczytajPrzedmiot(char *p);
};
```

```
class uczen : public osoba
{
  double srednia_ocen;
  int rok_rozpoczecia;
 public:
  uczen(char *n, char *i, int w, double s, int rr);
  ~uczen();
  double wypiszSredniaOcen();
  int wypiszRokRozpoczecia();
  void wczytajSredniaOcen(double s);
  void wczytajRokRozpoczecia(int rr);
};

class absolwent : public uczen
{
  int rok_zakonczenia;
 public:
  absolwent(char *n, char *i, int w, double s, int rr, int rz);
  ~absolwent();
  int wypiszRokZakonczenia();
  void wczytajRokZakonczenia(int z);
};
```

Konstruktor klasy pochodnej powinien obejmować pola własnej klasy oraz pola odziedziczone. Przykładem takiej funkcji jest przedstawiony poniżej konstruktor klasy `uczen`, który odziedziczone pola wczytuje poprzez konstruktor klasy nadrzędnej w hierarchii, czyli bazowej:

```
uczen::uczen(char *n, char *i, int w, double s, int rr):osoba(n,i,w)
{
 srednia_ocen = s;
 rok_rozpoczecia = rr;
}
```

Metody wczytujące i wypisujące wartości pól zawarte w klasie `uczen` mogą mieć następującą postać:

```
double uczen::wypiszSredniaOcen()
{
 return srednia_ocen;
}
```

```
void uczen::wczytajRokRozpoczecia(int rr)
{
 rok_rozpoczecia = rr;
}
```

Zadanie 3.50. Napisz program zawierający hierarchię klas przedstawioną w przykładzie 3.62. Zdefiniuj obiekty klas `uczen`, `absolwent` i `nauczyciel`. Zastosuj dynamiczny przydział pamięci.

3.9.4. Polimorfizm i metody wirtualne

Pola i metody typu obiektowego dziedziczone są w typach potomnych. Metody typu nadrzędnego mogą być zastępowane w typie potomnym nowymi metodami o tej samej nazwie, co nazywamy **pokrywaniem metody** dziedziczonej po przodku.

Definicja

Polimorfizm (czyli wielopostaciowość) polega na zdefiniowaniu metody o jednej nazwie w całej hierarchii klas. Wynika stąd, że w każdej klasie pojawi się metoda o tej samej nazwie. W języku C++ polimorfizm jest realizowany za pomocą **metod wirtualnych**, a obiekty z deklaracjami takich metod nazywamy **obiektami polimorficznymi**.

Deklaracja pewnej metody jako wirtualnej w definicji typu obiektowego następuje przez dodanie na początku słowa kluczowego `virtual`. Jeśli jakaś metoda jest zadeklarowana w typie nadrzędnym jako wirtualna, to wszystkie metody o takim samym identyfikatorze w typach potomnych muszą być także zadeklarowane jako wirtualne. W praktyce wystarczy deklaracja metody wirtualnej w klasie bazowej.

Przykład 3.63.

Przeanalizuj przedstawiony poniżej kod programu (*prog3_48.cpp*). Określ zdefiniowaną hierarchię klas. Wskaż metody wirtualne zastosowane w tym programie. Uzasadnij potrzebę ich wykorzystania.

```
#include <iostream>
using namespace std;
class pierwsza
{
 protected:
  int a;
  int b;
 public:
  pierwsza(int a, int b);
  ~pierwsza();
  virtual void oblicz1();
  virtual void oblicz2();
```

```
  int oblicz();
  int wypiszA();
  int wypiszB();
};
class druga: public pierwsza
{
  int c;
 public:
  druga(int a, int b, int c);
  ~druga();
  void oblicz1();
  void oblicz2();
  int wypiszC();
};
pierwsza::pierwsza(int aa, int bb): a(aa), b(bb)
{}
pierwsza::~pierwsza()
{}
void pierwsza::oblicz1()
{
 a*=2;
}
void pierwsza::oblicz2()
{
 b*=2;
}
int pierwsza::oblicz()
{
 oblicz1();
 oblicz2();
 return a+b;
}
druga::druga(int aa, int bb, int cc=3): pierwsza(aa, bb), c(cc)
{}
druga::~druga()
{}
void druga::oblicz1()
{
 a*=c;
}
```

```
void druga::oblicz2()
{
 b*=c;
}
int pierwsza::wypiszA()
{
 return a;
}
int pierwsza::wypiszB()
{
 return b;
}
int druga::wypiszC()
{
 return c;
}
int main()
{
 pierwsza *P = new pierwsza(2,3);
 druga *D;
 D = new druga(3,4);
 cout<<"klasa PIERWSZA"<<endl;
 cout<<"a="<<P->wypiszA()<<"\nb="<<P->wypiszB()<<endl;
 cout<<"wynik="<<P->oblicz()<<endl;
 cout<<"\nklasa DRUGA"<<endl;
 cout<<"a="<<D->wypiszA()<<"\nb="<<D->wypiszB()<<"\nc="
 <<D->wypiszC()<<endl;
 cout<<"wynik="<<D->oblicz()<<endl;
 delete P;
 delete D;
 return 0;
}
```

Po wykonaniu programu zostaną wypisane następujące komunikaty:

klasa PIERWSZA

a=2

b=3

wynik=10

klasa DRUGA

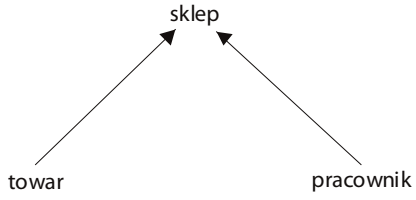
a=3

b=4

c=3

wynik=21

Zadanie 3.51. W pliku *prog3_49.cpp* znajduje się fragment programu zawierający hierarchię klas przedstawioną na rysunku 3.22.



Rysunek 3.22. Hierarchia klas do zadania 3.51

Uzupełnij kod tego programu, dodając podane elementy:

a) Zdefiniuj klasę `pracownik` składającą się z następujących elementów:

- pola:
  - `stanowisko` — stanowisko zajmowane przez pracownika,
  - `zasadnicze` — miesięczne wynagrodzenie zasadnicze jednego pracownika,
  - `premia` — miesięczna premia jednego pracownika,
- metody realizujące następujące działania:
  - wczytywanie wartości pól,
  - wypisanie wartości pól,
  - wypisanie nazwy i miesięcznych obrotów sklepu, jeśli liczba pracowników jest nie większa niż 5 i miesięczne obroty przekraczają 50 000 zł,
  - obliczenie całkowitego wynagrodzenia wszystkich pracowników z uwzględnieniem wynagrodzenia zasadniczego i premii.

b) Uzupełnij klasy `towar` i `sklep`, dodając do każdej z klas po jednej samodzielnie zdefiniowanej metodzie wykonującej operacje na polach.

c) Utwórz obiekt `T` klasy `towar` i obiekt `P` klasy `pracownik`.

W programie należy zastosować dynamiczne odwołanie do obiektów i następujący dostęp do składowych klas: pola prywatne, metody publiczne.