16

Mechanizm abstrakcji

Mechanizm abstrakcji to jedna z najważniejszych cech paradygmatu obiektowego. W ogólności polega on na ukrywaniu przed użytkownikiem szczegółów implementacji klas (obiektów), a pokazywaniu jedynie ich funkcjonalności. Innymi słowy, abstrakcja pozwala ukrywać to, w jaki sposób osiągnięto daną funkcjonalność, i skupić się jedynie na tej funkcjonalności, czyli na tym, co robi (wykonuje) dana metoda (obiekt).

W popularnych językach programowania, takich jak Java i C#, mechanizm abstrakcji jest implementowany za pomocą tzw. klas abstrakcyjnych i interfejsów.

W wymienionych językach programowania **klasa abstrakcyjna** (ang. *abstract class*) to klasa bazowa, która zawiera deklaracje (prototypy) tzw. metod abstrakcyjnych oraz kompletne definicje innych elementów członkowskich, np. zwykłych — instancyjnych zmiennych i funkcji członkowskich (metod). Przy czym **metoda abstrakcyjna** (ang. *abstract method*) zadeklarowana w bazowej klasie abstrakcyjnej to metoda, która musi zostać zdefiniowana w jej klasie pochodnej. Tym samym każda metoda abstrakcyjna jest metodą polimorficzną. Zadeklarowanie metody jako abstrakcyjnej automatycznie załącza mechanizm polimorfizmu dynamicznego, czyli polimorfizmu zachodzącego w czasie wykonywania programu.

Drugie z wymienionych wcześniej narzędzi abstrakcji w językach Java i C#, **interfejs** (ang. *interface*), jest strukturą programistyczną, która ukrywa szczegóły implementacji metod, a pokazuje jedynie ich zachowanie — funkcjonalność. Interfejsy — w przeciwieństwie do klas abstrakcyjnych — mogą zawierać wyłącznie deklaracje metod abstrakcyjnych, stałe i metody statyczne oraz metodę domyślną. W interfejsach nie można definiować zwykłych, instancyjnych zmiennych członkowskich i metod. Natomiast klasa abstrakcyjna może, co już zostało wspomniane, zawierać również np. zmienne i metody instancyjne. W językach Java i C# interfejsy można definiować niezależnie od klas abstrakcyjnych, jako zupełnie odrębne struktury programistyczne.

Zgodnie zaś z regułami dziedziczenia w Javie i C#, a szczególnie ze względu na brak możliwości dziedziczenia wielokrotnego, w wymienionych językach programowania dana klasa może dziedziczyć tylko po jednej klasie bazowej, np. klasie abstrakcyjnej, ale jednocześnie może dziedziczyć po wielu interfejsach.

Przy czym w języku C++ mechanizm abstrakcji może być realizowany na kilka sposobów. Jednym z najważniejszych jest użycie klas abstrakcyjnych. Ponadto abstrakcję można osiągnąć przez odpowiednie zorganizowanie struktury wewnętrznej klasy, jak też dzięki wykorzystaniu zbiorów nagłówkowych.

Interfejsy (znane z języków Java i C#) w C++ są implementowane wyłącznie za pomocą klas abstrakcyjnych. Taka możliwość wynika z ważnej, unikatowej cechy języka C++, polegającej na tym, że dziedziczenie może w nim być wielokrotne, czyli klasa pochodna może dziedziczyć po wielu klasach bazowych.

Biorąc pod uwagę powyższe, klasa abstrakcyjna w języku C++ łączy cechy klasy abstrakcyjnej i interfejsu z Javy i C#. Jak wspomniano wcześniej, w Javie i C# dana klasa może dziedziczyć po wielu interfejsach, ale tylko po jednej klasie bazowej (np. klasie abstrakcyjnej). W C++ określona klasa może dziedziczyć po wielu klasach, w tym klasach abstrakcyjnych.

16.1. Klasy abstrakcyjne

Jednym z najważniejszych sposobów uzyskania abstrakcji w języku C++ jest użycie klas abstrakcyjnych. W ogólności **klasa abstrakcyjna** (ang. *abstract class*) to klasa bazowa, która zawiera deklarację (prototyp) przynajmniej jednej **metody abstrakcyjnej** (ang. *abstract method*). W C++ metody abstrakcyjne implementuje się za pomocą **metod czysto wirtualnych** (ang. *pure virtual methods*).

Deklaracja metody czysto wirtualnej jest analogiczna do deklaracji zwykłej metody wirtualnej, z tym że deklaracja tej pierwszej jest zakończona przypisaniem do niej wartości 0. Innymi słowy, ciało funkcji czysto wirtualnej jest zastępowane przypisaniem `= 0`. Na przykład kod `virtual void wyswietlDane() = 0;` zawarty w definicji klasy abstrakcyjnej określa deklarację bezparametrowej metody czysto wirtualnej o nazwie `wyswietlDane`, niezwracającej na zewnątrz żadnej wartości.

UWAGA

W dalszej części podręcznika metody czysto wirtualne są nazywane metodami abstrakcyjnymi.

Metoda abstrakcyjna, której deklaracja (prototyp) jest zawarta w abstrakcyjnej klasie bazowej, powinna zostać zdefiniowana w jej klasie pochodnej. Jeśli jednak klasa pochodna, będąca bezpośrednim potomkiem — „dzieckiem" — abstrakcyjnej klasy bazowej, nie zawiera tej definicji, to mimo że jest klasą pochodną, jest traktowana jako klasa abstrakcyjna. W takim przypadku definicja metody abstrakcyjnej powinna się znaleźć w którymś z potomków na-

leżących do łańcucha dziedziczenia — np. „wnuku" klasy abstrakcyjnej, w której tę metodę zadeklarowano.

Jedną z najważniejszych cech klas abstrakcyjnych jest to, że nie można utworzyć obiektu, który byłby jej instancją. Przy czym dotyczy to zarówno abstrakcyjnych klas bazowych, czyli tych, które zawierają deklaracje metod abstrakcyjnych, jak i określonych klas pochodnych tych klas — tych, w których nie zdefiniowano metod abstrakcyjnych.

Abstrakcja implementowana w języku C++ za pomocą klas abstrakcyjnych jest ściśle powiązana z inną cechą paradygmatu obiektowego — polimorfizmem. Mianowicie zadeklarowanie funkcji członkowskiej klasy jako wirtualnej automatycznie załącza mechanizm polimorfizmu. Przy czym jest to polimorfizm dynamiczny, czyli polimorfizm zachodzący w czasie wykonywania programu.

UWAGA

Tematyka polimorfizmu dynamicznego została omówiona szczegółowo w podrozdziale 15.2 podręcznika.

Przykład 16.1

```
#include <iostream>
using namespace std;

// Definicja abstrakcyjnej klasy bazowej Osoba:
class Osoba {
public:
    // Deklaracje zmiennych członkowskich:
    string imie;
    string nazwisko;
    /* UWAGA
     * Klasa abstrakcyjna może zawierać deklaracje zwykłych — instancyjnych zmiennych i metod członkowskich.
     */
    // Deklaracja (prototyp) metody abstrakcyjnej — funkcji czysto wirtualnej:
    virtual void wyswietlDane() = 0;
};
// Definicja klasy pochodnej Pracownik:
class Pracownik : public Osoba {
public:
    string stanowisko;
    // Definicja metody abstrakcyjnej zadeklarowanej w bazowej klasie abstrakcyjnej:
    void wyswietlDane() {
        cout << imie << " " << nazwisko << ", " << stanowisko << endl;
    }
```

```
    /* UWAGA
     * Metoda abstrakcyjna zadeklarowana w bazowej klasie abstrakcyjnej powinna zostać zdefiniowana
     * w jej klasie pochodnej.
     */
};
// Definicja klasy pochodnej Uczen:
class Uczen : public Osoba {
public:
    string klasa;
    // Definicja metody abstrakcyjnej zadeklarowanej w bazowej klasie abstrakcyjnej:
    void wyswietlDane() {
        cout << imie << " " << nazwisko << ", " << klasa << endl;
    }
};

int main() {
    // Utworzenie obiektu pracownik jako instancji klasy pochodnej Pracownik:
    Pracownik pracownik;
    // Nadanie wartości zmiennym członkowskim obiektu pracownik:
    pracownik.imie = "Jan";
    pracownik.nazwisko = "Kowalski";
    pracownik.stanowisko = "nauczyciel";
    cout << "DANE PRACOWNIKA:" << endl;
    // Wywołanie funkcji wyswietlDane() zdefiniowanej w klasie pochodnej Pracownik:
    pracownik.wyswietlDane();

    // Utworzenie obiektu uczen jako instancji klasy pochodnej Uczen:
    Uczen uczen;
    // Nadanie wartości zmiennym członkowskim obiektu uczen:
    uczen.imie = "Maria";
    uczen.nazwisko = "Nowak";
    uczen.klasa = "4A";
    cout << "DANE UCZNIA:" << endl;
    // Wywołanie funkcji wyswietlDane() zdefiniowanej w klasie Uczen:
    uczen.wyswietlDane();

    return 0;
}
```

W programie zdefiniowano klasę abstrakcyjną `Osoba`, która jest klasą bazową dla dwóch klas pochodnych: `Pracownik` i `Uczen`. Definicja klasy `Osoba` zawiera deklarację metody abstrakcyjnej `wyswietlDane()`, która została zaimplementowana jako metoda czysto wirtualna. Oprócz tego w skład klasy `Osoba` wchodzą zmienne członkowskie `imie` i `nazwisko`.

Definicje obu klas pochodnych `Pracownik` i `Uczen` zawierają definicje metody abstrakcyjnej `wyswietlDane()`, którą zadeklarowano w klasie bazowej `Osoba`. Definicje te są od siebie w pełni niezależne.

Zadeklarowanie w klasie bazowej metody `wyswietlDane()` jako (czysto) wirtualnej powoduje automatyczne załączenie mechanizmu polimorfizmu dynamicznego, czyli polimorfizmu zachodzącego w czasie wykonywania programu. Metodę polimorficzną stanowi oczywiście wspomniana powyżej metoda `wyswietlDane()`. W wyniku tego klasy wchodzące w skład łańcucha dziedziczenia tworzą hierarchię polimorficzną.

Wykorzystano tu zatem trzy założenia (cechy) paradygmatu obiektowego: dziedziczenie, polimorfizm i abstrakcję.

Zaimplementowana w programie abstrakcja dotyczy klas, a nie danych. W szczególności abstrakcja obejmuje wspólny interfejs klas `Pracownik` i `Uczen`, którym jest metoda `wyswietlDane()`. Implementacja wymienionej metody na poziomie bazowej klasy abstrakcyjnej `Osoba` nie jest znana, ponieważ nie została tam określona. Jeśli spojrzeć z perspektywy klasy bazowej `Osoba`, wiadomy jest jedynie abstrakcyjny cel (zadanie) metody `wyswietlDane()`, czyli co ta metoda ma robić — ale nie wiadomo, w jaki sposób. Różne implementacje metody abstrakcyjnej `wyswietlDane()`, czyli opisanie, w jaki sposób osiągnięto jej założoną funkcjonalność, są zawarte w klasach pochodnych klasy abstrakcyjnej `Osoba`, tj. w klasach `Pracownik` i `Uczen`.

Ćwiczenie 16.1

Zmodyfikuj program z przykładu 16.1 — wszystkie zmienne członkowskie klas `Osoba`, `Pracownik` i `Uczen` zadeklaruj jako chronione (ang. `protected`). Zdefiniuj w wymienionych klasach publiczne metody dostępowe — settery i gettery odpowiadające każdej z tych zmiennych. Ponadto uzupełnij definicję klasy abstrakcyjnej `Osoba` i definicje klas pochodnych `Pracownik` i `Uczen` o definicje konstruktorów: domyślnych i parametrycznych.

Przykład 16.2

```
#include <iostream>
using namespace std;

// Definicja abstrakcyjnej klasy bazowej Osoba:
class Osoba {
public:
    string imie;
    string nazwisko;
    // Deklaracja metody abstrakcyjnej wyswietlDane():
    virtual void wyswietlDane() = 0; // funkcja czysto wirtualna
};
// Definicja abstrakcyjnej klasy bazowej Stanowisko:
class Stanowisko {
```

```
public:
    string stanowisko;
    // Deklaracja metody abstrakcyjnej wyswietlDane():
    virtual void wyswietlDane() = 0; // funkcja czysto wirtualna
};
// Definicja abstrakcyjnej klasy bazowej Klasa:
class Klasa {
public:
    string klasa;
    // Deklaracja metody abstrakcyjnej wyswietlDane():
    virtual void wyswietlDane() = 0; // funkcja czysto wirtualna
};
// Definicja klasy pochodnej Pracownik:
class Pracownik : public Osoba, public Stanowisko {
public:
    // Definicja metody wyswietlDane():
    void wyswietlDane() {
        cout << imie << " " << nazwisko << ", " << stanowisko << endl;
    }
    /* UWAGA
     * Metoda wyswietlDane() została zadeklarowana jako abstrakcyjna w klasach bazowych klasy Pracownik,
     * tj. klasach Osoba i Stanowisko.
     */
};
// Definicja klasy pochodnej Uczen:
class Uczen : public Osoba, public Klasa {
public:
    // Definicja metody wyswietlDane():
    void wyswietlDane() {
        cout << imie << " " << nazwisko << ", " << klasa << endl;
    }
    /* UWAGA
     * Metoda wyswietlDane() została zadeklarowana jako abstrakcyjna w klasach bazowych klasy Uczen,
     * tj. klasach Osoba i Klasa.
     */
};

int main() {
    Pracownik pracownik;
    pracownik.imie = "Jan";
    pracownik.nazwisko = "Kowalski";
    pracownik.stanowisko = "nauczyciel";
    cout << "DANE PRACOWNIKA:" << endl;
    pracownik.wyswietlDane();
```

```
    Uczen uczen;
    uczen.imie = "Maria";
    uczen.nazwisko = "Nowak";
    uczen.klasa = "4A";
    cout << "DANE UCZNIA:" << endl;
    uczen.wyswietlDane();

    return 0;
}
```

Funkcjonalność programu przedstawionego w tym przykładzie jest analogiczna do funkcjonalności programu z przykładu 16.1. Różnica pomiędzy tymi programami tkwi w zastosowanych w nich rodzajach dziedziczenia. Mianowicie w programie z przykładu 16.1 wykorzystano dziedziczenie hierarchiczne, a tutaj — dziedziczenie wielokrotne.

Metoda `wyswietlDane()` została zadeklarowana jako czysto wirtualna (czyli abstrakcyjna) w trzech klasach bazowych: `Osoba`, `Stanowisko` i `Klasa`. Wymienione klasy są zatem klasami abstrakcyjnymi. Założona w klasach bazowych funkcjonalność metody `wyswietlDane()` to wyświetlenie danych. Nie zostało tam jednak określone, w jaki sposób ma to być zrealizowane.

Definicje tej metody abstrakcyjnej `wyswietlDane()` są zawarte w klasach pochodnych `Pracownik` i `Uczen`. Należy zwrócić uwagę na to, że implementacje metody `wyswietlDane()` w wymienionych klasach pochodnych są odmienne. Stanowią one różne odpowiedzi na pytanie, w jaki sposób osiągnięto funkcjonalność tej metody założoną w klasach abstrakcyjnych.

Ćwiczenie 16.2

Porównaj szczegółowo kod programu z przykładu 16.2 z kodem programu z przykładu 16.1. Odpowiedz na następujące pytania:

- Które z zastosowanych rozwiązań jest bardziej przejrzyste?
- Które rozwiązanie umożliwia łatwiejszą rozbudowę o dodatkowe klasy `Nauczyciel` i `Wychowawca`? Załóż, że każdy nauczyciel jest pracownikiem, a każdy wychowawca — nauczycielem.

Uzasadnij udzielone odpowiedzi.

16.2. Interfejsy

Jak już wspomniano, **interfejs** (ang. *interface*) jest strukturą programistyczną, która ukrywa szczegóły implementacji funkcji członkowskich klas, czyli metod, a pokazuje jedynie ich funkcjonalność. Abstrakcja realizowana za pomocą interfejsów dotyczy zatem wyłącznie klas (metod) — nie danych.

W języku C++ interfejsy są implementowane wyłącznie za pomocą klas abstrakcyjnych będących klasami bazowymi dla innych klas, które wchodzą w skład łańcucha dziedziczenia. Taka możliwość wynika z unikatowej cechy języka C++ polegającej na tym, że można w nim wykorzystywać dziedziczenie wielokrotne, co oznacza, że dana klasa pochodna może dziedziczyć po wielu klasach bazowych.

W szczególności w języku C++ za interfejs można uważać klasę abstrakcyjną, która zawiera deklaracje metod abstrakcyjnych i ewentualnie definicje komponentów statycznych (np. metod statycznych). Jednocześnie zakłada się, że w skład interfejsu nie mogą wchodzić definicje instancyjnych elementów członkowskich, tj. zmiennych i metod instancyjnych.

Wykorzystanie interfejsów, jako jednych z ważniejszych narzędzi abstrakcji w programowaniu obiektowym, pozwala oddzielić wspólny interfejs klas wchodzących w skład łańcucha dziedziczenia od ich implementacji. Przy tym należy zwrócić uwagę, że ten wspomniany wspólny interfejs klas jest polimorficzny. Wynika to z właściwości metod abstrakcyjnych, które deklaruje się w języku C++ jako funkcje członkowskie czysto wirtualne. To z kolei pociąga za sobą automatyczne załączenie mechanizmu polimorfizmu dynamicznego.

Podsumowując, interfejs, jako abstrakcyjna klasa bazowa, może zawierać jedynie sygnatury, tj. deklaracje (prototypy) metod. Są to metody abstrakcyjne. Implementacje metod abstrakcyjnych zadeklarowanych w interfejsie zaś są definiowane w jego klasach pochodnych — zgodnie z zasadami polimorfizmu dynamicznego.

W polimorficznej hierarchii klas wchodzących w skład łańcucha dziedziczenia interfejsy są podnoszone (ang. *hoisting*) na samą górę tej hierarchii. Zapewnia to efektywne i przejrzyste oddzielenie interfejsu klas polimorficznych od ich implementacji. Klasa interfejsu wyraża jedynie (abstrakcyjne) możliwości, cele i zadania, natomiast ich konkretna implementacja nie jest w tym miejscu istotna. Za wspomnianą implementację odpowiadają klasy pochodne interfejsu. Dzięki temu implementacje można dodawać/zmieniać w zależności od potrzeb — bez konieczności zmiany istniejącego kodu interfejsu.

Przykład 16.3

```
#include <iostream>
using namespace std;

// Definicja interfejsu Info:
class Info {
public:
    // Deklaracja metody abstrakcyjnej:
    virtual void wyswietlDane() = 0;
};
/* UWAGA
 * Klasa abstrakcyjna — interfejs Info zawiera wyłącznie deklarację metody abstrakcyjnej wyswietlDane(),
 * natomiast nie zawiera definicji żadnych innych zmiennych ani funkcji członkowskich.
 */
```

```
// Definicja klasy bazowej Osoba:
class Osoba {
public:
    string imie;
    string nazwisko;
    string szkola;
};
// Definicja klasy pochodnej Pracownik:
class Pracownik : public Info, public Osoba {
    /* UWAGA
     * Klasa pochodna Pracownik dziedziczy po interfejsie Info oraz klasie Osoba. Jest to dziedziczenie wielokrotne.
     */
public:
    // Deklaracja zmiennej członkowskiej:
    string stanowisko;
    // Definicja metody wyswietlDane() zadeklarowanej w interfejsie Info jako metoda abstrakcyjna:
    void wyswietlDane() {
        cout << "Imię: " << this->imie << endl;
        cout << "Nazwisko: " << this->nazwisko << endl;
        cout << "Stanowisko: " << this->stanowisko << endl;
    }
};
// Definicja klasy pochodnej Uczen:
class Uczen : public Info, public Osoba {
    /* UWAGA
     * Klasa pochodna Uczen dziedziczy po interfejsie Info i klasie Osoba.
     */
public:
    string klasa;
    // Definicja metody wyswietlDane() zadeklarowanej w interfejsie Info jako metoda abstrakcyjna:
    void wyswietlDane() {
        cout << "Imię: " << this->imie << endl;
        cout << "Nazwisko: " << this->nazwisko << endl;
        cout << "Klasa: " << this->klasa << endl;
    }
};

int main() {
    // Utworzenie obiektu pracownik jako instancji klasy pochodnej Pracownik:
    Pracownik pracownik;
    // Nadanie wartości początkowych zmiennym instancyjnym obiektu pracownik:
    pracownik.imie = "Jan";
    pracownik.nazwisko = "Kowalski";
    pracownik.stanowisko = "nauczyciel";
```

```
    cout << "DANE PRACOWNIKA:" << endl;
    // Wywołanie metody wyswietlDane() zdefiniowanej w klasie Pracownik:
    pracownik.wyswietlDane();
    cout << endl;

    Uczen uczen;
    uczen.imie = "Maria";
    uczen.nazwisko = "Nowak";
    uczen.klasa = "4A";
    cout << "DANE UCZNIA:" << endl;
    uczen.wyswietlDane();

    return 0;
}
```

W programie zaimplementowano mechanizm abstrakcji dotyczący klas. W szczególności zdefiniowano tutaj klasę abstrakcyjną `Info`. Należy zwrócić uwagę, że w jej skład nie wchodzą żadne instancyjne zmienne i metody członkowskie, lecz jedynie deklaracja metody abstrakcyjnej `wyswietlDane()`. Dlatego też klasa `Info` odpowiada strukturze programistycznej nazywanej interfejsem — znanej z takich języków programowania jak Java i C#.

W językach Java i C# dana klasa może dziedziczyć po wielu interfejsach, ale tylko po jednej klasie bazowej. Natomiast w C++ można zaimplementować dziedziczenie wielokrotne — jak w omawianym programie. Mianowicie klasy pochodne `Pracownik` i `Uczen` dziedziczą po tych samych dwóch klasach bazowych: klasie abstrakcyjnej (interfejsie) `Info` i zwykłej klasie `Osoba`.

W klasach `Pracownik` i `Uczen` zdefiniowano metodę `wyswietlDane()`, zadeklarowaną w interfejsie `Info` jako metoda abstrakcyjna. Funkcjonalność i implementacja tej metody w klasach `Pracownik` i `Uczen` są odmienne. Tym samym, jeśli spojrzeć z perspektywy interfejsu `Info`, nastąpiło oddzielenie wspólnego interfejsu klas `Pracownik` i `Uczen`, wchodzących w skład łańcucha dziedziczenia, od ich implementacji, która jest w nich zawarta.

Na poziomie interfejsu `Info` implementacja metody abstrakcyjnej `wyswietlDane()` nie jest znana, ponieważ nie została tam określona. Wiadome jest jedynie jej zadanie ogólne, którym jest odpowiedź na pytanie: co robi (wykonuje) ta metoda?

Jak wspomniano wcześniej, wspólny interfejs klas pochodnych `Pracownik` i `Uczen`, obejmujący metodę `wyswietlDane()`, został oddzielony od jej implementacji, która jest zawarta w wymienionych klasach pochodnych. Tym samym ewentualna modyfikacja kodu metody `wyswietlDane()` nie oznacza konieczności zmiany jej interfejsu zawartego w klasie (interfejsie) `Info`. Należy podkreślić, że interfejs `Info` został „podniesiony" na samą górę hierarchii klas wchodzących w skład łańcucha dziedziczenia.

Inną ważną cechą zaimplementowanej w programie hierarchii klas jest jej polimorficzność. Dotyczy to oczywiście polimorfizmu dynamicznego, czyli polimorfizmu zachodzącego w cza-

sie wykonywania programu. Wynika to z unikatowej cechy języka C++, w którym metoda abstrakcyjna `wyswietlDane()` została zadeklarowana w klasie abstrakcyjnej, interfejsie `Info`, jako funkcja (czysto) wirtualna.

Ćwiczenie 16.3

Zmodyfikuj program zawarty w przykładzie 16.3 — zdefiniuj dodatkowy interfejs o nazwie `InfoGminaSzkola` zawierający definicje dwóch metod statycznych, które pozwolą wyświetlić na ekranie nazwę szkoły oraz prowadzącego je miasta (gminy). Wykorzystaj interfejs `InfoGminaSzkola` w definicjach klas `Pracownik` i `Uczen`.

16.3. Abstrakcja danych

W programowaniu obiektowym stosowanie abstrakcji oznacza udostępnianie światu zewnętrznemu tylko wybranych informacji, natomiast inne elementy, np. szczegóły implementacji, są ukrywane. Była już o tym mowa wcześniej w tym rozdziale w odniesieniu do klas abstrakcyjnych i interfejsów. Metody wirtualne, które można deklarować w klasach jako abstrakcyjne, są metodami „bez ciała". Są zatem „bytami nieistniejącymi", abstrakcyjnymi — to jedynie określenie celu, zadania, czyli operacji do wykonania.

Z punktu widzenia innych elementów członkowskich klasy pojęcie abstrakcji może dotyczyć również danych, które są w niej przechowywane w jej instancjach — obiektach. W szczególności **abstrakcja danych** (ang. *data abstraction*) dotyczy danych reprezentowanych w klasie (obiekcie) przez zmienne członkowskie. Idea abstrakcji danych sprowadza się w praktyce do ukrycia przed światem zewnętrznym szczegółów implementacji danych (np. ukrycia typów zmiennych członkowskich), a udostępnienia w otoczeniu klasy jedynie interfejsu tych danych (np. identyfikatorów zmiennych członkowskich).

W języku C++ abstrakcję danych można uzyskać dzięki odpowiedniej organizacji i budowie klas. Proces ten jest wspomagany przez użycie specyfikatorów dostępu do ich elementów członkowskich (ang. *members*): `private`, `protected` i `public`. Specyfikatory `private` i `protected` zapewniają ukrycie szczegółów implementacji klasy (np. danych) przed kodem z otoczenia klasy. Jeżeli trzeba zapewnić niejawność implementacji danych pojedynczej klasy, najwłaściwszym rozwiązaniem jest użycie specyfikatora `private`. Konieczność ukrycia implementacji danych w obrębie łańcucha dziedziczenia w hierarchii klas wymusza zastosowanie specyfikatora `protected`. Specyfikator `public` zaś jest używany w odniesieniu do metod dostępowych, które zapewniają obsługę interfejsu danych. Metody te odpowiadają za funkcjonalność obiektu (obiektów) oraz manipulowanie danymi.

Abstrakcja danych jest ściśle powiązana z inną zasadą programowania obiektowego, mechanizmem hermetyzacji — ukrywania danych. W praktyce hermetyzacja danych zazwyczaj jest realizowana przy użyciu setterów i getterów, które są publicznymi metodami dostępowymi do prywatnych/chronionych zmiennych członkowskich klasy.

UWAGA

Tematyka hermetyzacji — ukrywania danych — została omówiona w podrozdziale 13.2.

Niewątpliwą zaletą stosowania zasad abstrakcji danych jest skuteczna ochrona implementacji tych danych. Ma to ogromne znaczenie np. w razie wystąpienia przypadkowych błędów na poziomie użytkownika (programisty), które mogą doprowadzić do niekontrolowanego stanu obiektu. Ponadto oddzielenie implementacji danych od ich interfejsu zapewnia łatwość modyfikowania klasy. Mianowicie nawet jeśli implementacja danych się zmieni, wystarczy przeprowadzić analizę kodu ich interfejsu, tj. kodu publicznych metod dostępowych, i w razie potrzeby wprowadzić niezbędne korekty. Ponadto w niektórych sytuacjach jest niepożądane, by użytkownik (tj. otoczenie klasy) miał wiedzę dotyczącą implementacji klasy (niekiedy jest to wręcz szkodliwe).

Przykład 16.4

```
#include <iostream>
using namespace std;

// Definicja klasy Pracownik:
class Pracownik {
private:
    // Deklaracje prywatnych zmiennych członkowskich:
    string imie, nazwisko;
    string stanowisko;
    int nr_legit;
    /* UWAGA
     * Szczegóły implementacji danych pracownika przechowywanych w zmiennych członkowskich klasy
     * nie są dostępne dla świata zewnętrznego — są ukryte i niedostępne bezpośrednio dla kodu w otoczeniu klasy.
     */
public:
    // Definicje publicznych setterów i getterów:
    void setImie(string pImie) {
        this->imie = pImie;
    }
    string getImie() {
        return this->imie;
    }
    void setNazwisko(string pNazwisko) {
        this->nazwisko = pNazwisko;
    }
    string getNazwisko() {
        return this->nazwisko;
    }
    void setStanowisko(string pStanowisko) {
```

```
        this->stanowisko = pStanowisko;
    }
    string getStanowisko() {
        return this->stanowisko;
    }
    void setNrLegit(int pNrLegit) {
        const int temp = 1000;
        this->nr_legit = pNrLegit + temp;
    }
    int getNrLegit() {
        return this->nr_legit;
    }
    /* UWAGA
     * Settery i gettery stanowią publiczny interfejs danych pracownika, odpowiadających prywatnym
     * zmiennym członkowskim klasy.
     */
};
// Definicja zwykłej funkcji niezależnej od klasy Pracownik:
void wyswietlDane(Pracownik pPracownik) {
    cout << "Imię: " << pPracownik.getImie() << endl;
    cout << "Nazwisko: " << pPracownik.getNazwisko() << endl;
    cout << "Stanowisko: " << pPracownik.getStanowisko() << endl;
    cout << "Numer legitymacji: " << pPracownik.getNrLegit() << endl;
}

int main() {
    // Utworzenie obiektu pracownik jako instancji klasy Pracownik:
    Pracownik pracownik;
    // Nadanie wartości początkowych zmiennym członkowskim obiektu pracownik:
    pracownik.setImie("Jan");
    pracownik.setNazwisko("Kowalski");
    pracownik.setStanowisko("nauczyciel");
    pracownik.setNrLegit(12);
    /* UWAGA
     * Ustalenie/zmiana danych pracownika jest realizowana za pomocą publicznych metod dostępowych — setterów.
     * Metody te stanowią interfejs wspomnianych danych, które, jako prywatne, nie są dostępne w sposób bezpośredni
     * w otoczeniu klasy.
     */
    // Prezentacja danych pracownika na ekranie monitora:
    cout << "DANE PRACOWNIKA:" << endl;
    // Wywołanie funkcji wyswietlDane(), niezależnej od klasy Pracownik:
    wyswietlDane(pracownik);

    return 0;
}
```

W programie zademonstrowano implementację abstrakcji danych pracownika, reprezentowanych przez zmienne członkowskie: `imie`, `nazwisko`, `stanowisko`, `nr_legit` obiektu `pracownik`. Obiekt `pracownik` jest instancją klasy `Pracownik`.

Szczegóły implementacji tych danych są ukryte przed światem zewnętrznym dzięki zadeklarowaniu ich w klasie `Pracownik` przy użyciu specyfikatora dostępu `private`. Interfejs ukrytych danych zapewniają publiczne metody dostępowe, czyli settery i gettery, poszczególnych zmiennych członkowskich oraz metoda instancyjna `wyswietlDane()`.

Implementacja danych w klasie `Pracownik` została skutecznie oddzielona od ich interfejsu. Oznacza to, że np. dodanie nowych elementów członkowskich do klasy `Pracownik`, takich jak zmienna członkowska odpowiadająca stażowi pracy pracownika, nie będzie miało żadnego wpływu na dotychczasowy interfejs danych w tej klasie. Jednocześnie uwzględnienie nowych danych pracownika zaimplementowanych w klasie `Pracownik` sprawi, że będzie trzeba zmienić kod funkcji `wyswietlDane()`, której zadaniem jest wyświetlenie na ekranie monitora wszystkich danych pracownika.

Ćwiczenie 16.4

Zmodyfikuj program zawarty w przykładzie 16.4 — uwzględnij w klasie `Pracownik` dodatkowe dane pracownika: staż pracy oraz wynagrodzenie podstawowe. Zaimplementuj abstrakcję nowych danych.

16.4. Mechanizm abstrakcji a pliki nagłówkowe

W języku C++ założenia abstrakcji można zrealizować również przez wykorzystanie plików nagłówkowych (ang. *header files*). W tym celu kod źródłowy zawierający określone elementy składowe definicji klasy można umieścić w plikach zewnętrznych:

- pliku nagłówkowym (*.h*),
- pliku z kodem źródłowym (*.cpp*).

Plik nagłówkowy zawiera definicję klasy. Przy czym w definicji klasy zamieszcza się jedynie jej interfejs, tj. deklaracje (a nie definicje) publicznych metod dostępowych do prywatnych (lub chronionych) zmiennych członkowskich. Kompletne definicje wspomnianych metod publicznych klasy, czyli ich implementacja, powinny być zawarte w pliku z kodem źródłowym *.cpp*.

UWAGA

Wykorzystanie w programie zbiorów nagłówkowych zostało omówione wcześniej — w podrozdziale 9.1.

Zgodnie z przedstawionymi wcześniej założeniami zarówno implementacja danych reprezentowanych przez prywatne (lub chronione) zmienne członkowskie klasy, jak i implementacja publicznych metod dostępowych do tych danych zostają ukryte przed światem zewnętrznym. Rolę „kontenera" (pojemnika) przeznaczonego na ukryte szczegóły implementacji funkcji członkowskich klasy odgrywa plik nagłówkowy. Jednocześnie należy zwrócić uwagę, że jest to kontener zewnętrzny. Kontenerem wewnętrznym jest klasa „sama w sobie", w której zmienne członkowskie zostały zadeklarowane jako prywatne lub chronione. Mamy więc w tym przypadku do czynienia z abstrakcją dotyczącą zarówno danych, jak i innych komponentów klasy — w szczególności publicznych metod instancyjnych.

Implementacja hermetyzacji — ukrycia danych — zrealizowana na poziomie klasy z jednoczesnym wykorzystaniem plików nagłówkowych w celu ukrycia przed światem zewnętrznym szczegółów implementacji jej metod publicznych skutkuje efektywnym oddzieleniem implementacji klasy od jej interfejsu. Dzięki temu mechanizm abstrakcji dotyczy wszystkich komponentów składowych klasy.

Przykład 16.5

Program główny:

```
#include <iostream>
// Dołączenie do aplikacji zdefiniowanego pliku nagłówkowego:
#include "kolo.h"
/* UWAGA
 * Kompilator będzie poszukiwał pliku nagłówkowego w katalogu bieżącym (czyli tym, w którym jest zapisany plik main.cpp).
 */
using namespace std;

int main() {
    // Utworzenie obiektu kolo klasy Kolo:
    Kolo kolo(1); // niejawne wywołanie konstruktora parametrycznego z argumentem 1
    // Odczyt bieżącej wartości promienia:
    cout << "Promień koła: " << kolo.getPromien() << endl;
    /* UWAGA
     * Odczyt wartości prywatnej zmiennej członkowskiej _r został zrealizowany z użyciem gettera getPromien().
     */
    // Obliczenie pola i obwodu koła:
    double poleK = kolo.pole();
    double obwodK = kolo.obwod();
    /* UWAGA
     * Obliczenie pola i obwodu koła zostało zrealizowane z zastosowaniem publicznych metod instancyjnych klasy Kolo.
     */
    // Prezentacja wyników na ekranie monitora:
    cout << "Pole koła: " << poleK << endl;
    cout << "Obwód koła: " << obwodK << endl;
```

```
    return 0;
}
```

Zawartość pliku nagłówkowego *kolo.h*:

```
// PLIK NAGŁÓWKOWY
// Deklaracja klasy Kolo:
class Kolo {
private:
    // Deklaracja prywatnej zmiennej członkowskiej:
    double _r;
public:
    // Prototypy konstruktorów:
    Kolo();
    Kolo(double);
    // Deklaracje publicznych metod dostępowych:
    void setPromien(double); // setter
    double getPromien(); // getter
     // Prototypy (nagłówki) zwykłych metod instancyjnych:
     double pole();
     double obwod();
};
```

Zawartość pliku *kolo.cpp*:

```
// PLIK Z KODEM ŹRÓDŁOWYM (IMPLEMENTACJĄ) METOD PUBLICZNYCH
#define _USE_MATH_DEFINES // w celu użycia stałej M_PI, która nie jest zdefiniowana w standardowym C/C++
#include <cmath>
#include "kolo.h"
// Definicje publicznych metod członkowskich klasy Kolo
// Definicje konstruktorów:
Kolo::Kolo() {}
Kolo::Kolo(double r) {
    _r = r;
}
// Definicja settera:
void Kolo::setPromien(double r)    {
    _r = r;
}
// Definicja gettera:
double Kolo::getPromien() {
    return _r;
}
// Definicje zwykłych metod instancyjnych:
double Kolo::pole() {
    return M_PI * _r * _r;
}
double Kolo::obwod() {
    return 2 * M_PI *_r;
}
```

Implementacja danych reprezentowanych w klasie `Kolo` przez prywatną zmienną członkowską `_r` została oddzielona od jej bezpośredniego, publicznego interfejsu — metod dostępowych: settera `setPromien()` i gettera `getPromien()`. Mamy więc tutaj do czynienia z abstrakcją danych zrealizowaną za pomocą klasy `Kolo`, zorganizowanej wewnętrznie z użyciem specyfikatorów dostępu — odpowiednio: `private` i `public` — do jej elementów członkowskich.

Oprócz tego wykorzystano tutaj plik nagłówkowy *kolo.h* oraz plik z kodem źródłowym *kolo.cpp*. W podanym pliku nagłówkowym ukryto przed światem zewnętrznym szczegóły implementacji klasy `Kolo`. W szczególności dotyczy to implementacji publicznych metod instancyjnych należących do klasy `Kolo`, w tym publicznych metod dostępowych do prywatnej zmiennej członkowskiej `_r` — settera `setPromien()` i gettera `getPromien()`. Kompletne definicje wspomnianych metod publicznych są zawarte w pliku *kolo.cpp*, którego zawartość nie jest udostępniana użytkownikowi w sposób bezpośredni.

Tak więc mechanizm abstrakcji dotyczy w zaprezentowanym programie wszystkich komponentów klasy `Kolo` — zarówno prywatnych danych, jak i publicznych metod członkowskich:

- implementacja klasy `Kolo` została oddzielona od jej interfejsu,
- szczegóły implementacji klasy `Kolo` zostały ukryte przed światem zewnętrznym.

Ćwiczenie 16.5

Zmodyfikuj program zawarty w przykładzie 16.5 — zamiast klasy `Kolo` zdefiniuj klasę `Prostokat` zawierającą elementy członkowskie, które umożliwią ustalenie wartości i odczyt parametrów prostokąta oraz obliczenie jego pola i obwodu. Zaimplementuj mechanizm abstrakcji przy użyciu zbioru nagłówkowego *prostokat.h*. Kody źródłowe definicji publicznych funkcji członkowskich klasy `Prostokat` zapisz w pliku *prostokat.cpp*. Abstrakcję danych odpowiadających parametrom prostokąta zrealizuj przez odpowiednie wewnętrzne zorganizowanie klasy `Prostokat`.

16.5. Pytania i zadania kontrolne

16.5.1. Pytania

1. Jaką rolę w programowaniu obiektowym odgrywają klasy abstrakcyjne? W jaki sposób definiuje się klasy abstrakcyjne w języku C++?
2. Co to jest metoda abstrakcyjna? W jaki sposób definiuje się metody abstrakcyjne w C++?
3. Czym różni się klasa abstrakcyjna od interfejsu w języku C++?
4. Czy mechanizm abstrakcji jest związany wyłącznie z danymi, czy może dotyczyć również innych elementów członkowskich (komponentów) klasy?
5. Na czym polega mechanizm abstrakcji danych?
6. Czy wykorzystanie zbiorów nagłówkowych może być pomocne w implementacji mechanizmu abstrakcji?

16.5.2. Zadania

1. Napisz program pozwalający przetwarzać dane pracowników szpitala, a dokładniej: lekarzy i ordynatorów oddziałów szpitalnych. Załóż, że każdy lekarz ma określoną specjalizację (np. kardiologia), a każdy ordynator jest lekarzem i jednocześnie kieruje pojedynczym oddziałem w szpitalu (np. oddziałem wewnętrznym). Zastosuj założenia dziedziczenia i abstrakcji. Mechanizm abstrakcji zrealizuj przy użyciu klas abstrakcyjnych. Wykonaj testy działania programu na danych kilku lekarzy i ordynatorów.
2. Zrób tak jak w zadaniu 1., lecz mechanizm abstrakcji zrealizuj za pomocą interfejsu (interfejsów).
3. Napisz program pozwalający przetwarzać dane wybranych pracowników komendy policji. Uwzględnij policjantów oraz naczelników wydziałów. Załóż, że każdy policjant ma określony stopień służbowy (np. komisarz), a każdy naczelnik jest policjantem i jednocześnie kieruje jednym wydziałem w komendzie (np. wydziałem ruchu drogowego). Wykorzystaj mechanizmy dziedziczenia i abstrakcji. Mechanizm abstrakcji zrealizuj przy użyciu klas abstrakcyjnych. Wykonaj testy działania programu na danych kilku policjantów i naczelników wydziałów.
4. Zrób tak jak w zadaniu 3., lecz mechanizm abstrakcji zrealizuj za pomocą interfejsu (interfejsów).
5. Napisz program pozwalający obliczyć objętość, pole powierzchni całkowitej oraz długość wszystkich krawędzi prostopadłościanu. Dane wejściowe (parametry prostokąta) mają być wprowadzane z klawiatury, a wyniki wyświetlane na ekranie monitora. Wykorzystaj mechanizm abstrakcji danych.
6. Zrób tak jak w zadaniu 5., z tym że dodatkowo uwzględnij abstrakcję uzyskaną dzięki zastosowaniu pliku nagłówkowego *prostopadloscian.h* oraz pliku z kodem źródłowym *prostopadloscian.cpp*. Plik *prostopadloscian.h* powinien zawierać definicję klasy `Prostopadloscian` z deklaracjami prywatnych zmiennych i publicznych metod członkowskich, a plik *prostopadloscian.cpp* — kompletne definicje publicznych metod członkowskich klasy `Prostopadloscian`.