WYDANIE III

*Przewodnik dla poszukujących pracy!*

# Programista szuka pracy

## *Kulisy rekrutacji w branży IT*

John Mongan, Noah Kindler, Eric Giguere

Helion

Tytuł oryginału: Programming Interviews Exposed: Secrets to Landing Your Next Job, Third Edition

Tłumaczenie: Łukasz Piwko

ISBN: 978-83-246-9861-5



Wydawnictwo HELION
ul. Kościuszki 1c, 44-100 GLIWICE
tel. 32 231 22 19, 32 230 98 63
e-mail: *helion@helion.pl*
WWW: *http://helion.pl* (księgarnia internetowa, katalog książek)

Drogi Czytelniku!
Jeżeli chcesz ocenić tę książkę, zajrzyj pod adres
*http://helion.pl/user/opinie/rokwpr*
Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

Printed in Poland.

# Spis treści

# 10

# Programowanie obiektowe

Większość profesjonalnych programistów używa obiektowych języków programowania, takich jak Java, C# czy C++. Nawet język JavaScript, mimo że nie jest obiektowy, ma pewne cechy obiektowych języków programowania pod postacią obiektów prototypowych i sprytnego wykorzystania definicji funkcji. Dlatego każdy kandydat na rozmowie kwalifikacyjnej powinien dobrze rozumieć zasady programowania obiektowego.

## PODSTAWY

Korzenie obiektowych technik programowania sięgają kilkadziesiąt lat wstecz, do takich języków jak Simula i Smalltalk. Programowanie obiektowe stało się tematem wielu opracowań naukowych od czasu, gdy aktywni programiści zaczęli powszechnie z niego korzystać.

### Klasy i obiekty

Istnieje wiele sposobów i nie ma pełnej zgody co do tego, jak opisać i zdefiniować obiektowość jako technikę programowania, ale wszyscy zgodnie mówią o klasach i obiektach. **Klasa** to abstrakcyjna definicja czegoś, co ma **atrybuty** (czasami nazywane **własnościami** lub **stanami**) i wzorce zachowań (**możliwości** lub **metody**). **Obiekt** to konkretny egzemplarz klasy, który ma własny stan niepowiązany ze stanami innych egzemplarzy tej klasy. Poniżej znajduje się definicja klasy punktu będącego parą liczb całkowitych reprezentujących wartości *x* i *y* w kartezjańskim układzie współrzędnych.

```
public class Point {
    private int x;
    private int y;
    public Point( int x, int y ){
        this.x = x;
        this.y = y;
    }
    public Point( Point other ){
        x = other.getX();
        y = other.getY();
    }
```

```
    public int getX(){ return x; }
    public int getY(){ return y; }
    public Point relativeTo( int dx, int dy ){
        return new Point( x + dx, y + dy );
    }
    public String toString(){
        StringBuffer b = new StringBuffer();
        b.append( '(' );
        b.append( x );
        b.append( ',' );
        b.append( y );
        b.append( ')' );
        return b.toString();
    }
}
```

Aby reprezentować dowolny punkt, należy utworzyć egzemplarz klasy `Point` przy użyciu odpowiednich wartości.

```
Point p1 = new Point( 5, 10 );
Point p2 = p1.relativeTo( -5, 5 );
System.out.println( p2.toString() ); // Drukuje (0,15).
```

Ten prosty przykład jest ilustracją jednej z podstawowych zasad programowania obiektowego — **hermetyzacji**, czyli ukrywania szczegółów implementacji.

## Dziedziczenie i polimorfizm

Dwa inne, bardzo ważne pojęcia obiektowości to dziedziczenie i polimorfizm, które są ściśle spokrewnione. **Dziedziczenie** polega na tym, że można zdefiniować klasę jako zmodyfikowaną lub bardziej wyspecjalizowaną wersję innej klasy. Gdy klasa B dziedziczy po klasie A (w niektórych językach programowania, np. Javie, oznacza się to słowem kluczowym `extends`), klasa A jest **rodzicem** klasy B lub jej **klasą bazową** albo **nadklasą**, natomiast klasa B jest **podklasą** lub **klasą podrzędną** klasy A. Wszystkie zachowania zdefiniowane w klasie A znajdują się też w klasie B, chociaż mogą być nieco zmienione. Egzemplarza klasy B można nawet użyć wszędzie tam, gdzie można zastosować egzemplarz klasy A.

**Polimorfizm** to możliwość dostarczania wielu implementacji zachowania i wybierania jednej z nich na podstawie kontekstu. Przykładowo klasa może zawierać definicje dwóch wersji jednej metody, z których każda przyjmuje inne parametry wywołania. Ta sama metoda może też być zdefiniowana zarówno w nadklasie, jak i podklasie, przy czym ta druga *przesłania* pierwszą w przypadku wywołań na obiektach podklasy. Wybór metody do wywołania może odbywać się podczas kompilacji kodu lub działania programu.

Klasycznym przykładem przywoływanym przy objaśnianiu dziedziczenia i polimorfizmu jest implementacja biblioteki kształtów geometrycznych używanej w aplikacji do rysowania wektorowego. Na szczycie hierarchii znajduje się klasa o nazwie `Shape` zawierająca definicje cech wspólnych wszystkich kształtów.

```
public abstract class Shape {
    protected Point center;
    protected Shape( Point center ){
        this.center = center;
    }
    public Point getCenter(){
        return center; // Ponieważ punkt jest niezmienny.
    }
```

```
    public abstract Rectangle getBounds();
    public abstract void draw( Graphics g );
}
```

Następnie można zdefiniować specjalne podklasy `Rectangle` (prostokąt) i `Ellipse` (elipsa).

```
public class Rectangle extends Shape {
    private int h;
    private int w;

    public Rectangle( Point center, int w, int h ){
        super( center );
        this.w = w;
        this.h = h;
    }
    public Rectangle getBounds(){
        return this;
    }
    public int getHeight(){ return h; }
    public int getWidth(){ return w; }
    public void draw( Graphics g ){
        .... // Kod rysujący prostokąt.
    }
}
public class Ellipse extends Shape {
    private int a;
    private int b;
    public Ellipse( Point center, int a, int b ){
        super( center );
        this.a = a;
        this.b = b;
    }
    public Rectangle getBounds(){
        return new Rectangle( center, a * 2, b * 2 );
    }
    public int getSemiMajorAxis(){ return a; }
    public int getSemiMinorAxis(){ return b; }
    public void draw( Graphics g ){
        .... // Kod rysujący elipsę.
    }
}
```

Klasy `Rectangle` i `Ellipse` można jeszcze dalej wyspecjalizować przez utworzenie ich podklas `Square` (kwadrat) i `Circle` (koło).

Mimo że w bibliotece może być zdefiniowanych wiele kształtów, część aplikacji rysująca te figury nie jest skomplikowana.

```
void paintShapes( Graphics g, List<Shape> shapes ){
    for( Shape s : shapes ){
        s.draw( g );
    }
}
```

Aby dodać nowy kształt do biblioteki, wystarczy utworzyć podklasę jednej z istniejących klas i zaimplementować w niej różnice.

# KONSTRUKCJA I DESTRUKCJA OBIEKTÓW

Obiekty są egzemplarzami klas. Tworzenie obiektu nazywa się **konstrukcją obiektu**. W procesie tym wywołuje się metodę klasy zwaną **konstruktorem**. Konstruktor inicjuje stan obiektu,

co zazwyczaj wymaga wywołania (jawnie lub niejawnie) konstruktorów klas nadrzędnych, aby te zainicjowały swoją część stanu obiektu.

Niszczenie obiektów jest trochę bardziej skomplikowane od ich tworzenia. W języku C++ wywołuje się w tym celu funkcję zwaną **destruktorem**, której zadaniem jest skasowanie stanu obiektu. Destruktory są wywoływane automatycznie, gdy obiekt wychodzi poza zakres dostępności lub w wyniku użycia operatora `delete` służącego do usuwania dynamicznie utworzonych obiektów — ważne jest zapamiętywanie obiektów, aby uniknąć wycieków pamięci. Natomiast w językach, takich jak C# i Java, nieużywane obiekty znajduje i usuwa mechanizm zwany **śmieciarką** (ang. *garbage collector*). W takim przypadku czas i miejsce (często dzieje się to w osobnym wątku zdefiniowanym przez system) destrukcji są poza kontrolą aplikacji. Opcjonalnie system może wywołać metodę *finalizującą* przed destrukcją obiektu, aby dać mu szansę na samooczyszczenie, zanim go ostatecznie zniszczy. (W językach C# i Java obiekt może się „wskrzesić" w finalizatorze).

# PROBLEMY DOTYCZĄCE PROGRAMOWANIA OBIEKTOWEGO

Zadania związane z programowaniem obiektowym najczęściej skupiają się na samych pojęciach obiektowości, zwłaszcza na kwestiach dotyczących języków programowania używanych w firmie.

## Interfejsy i klasy abstrakcyjne

> **PROBLEM** Czym różni się interfejs od klasy abstrakcyjnej w programowaniu obiektowym?

Odpowiedź na to pytanie zależy od języka programowania, ale można przedstawić ogólne definicje.

- **Interfejs** zawiera deklaracje zestawu powiązanych metod, poza jakąkolwiek klasą.
- **Klasa abstrakcyjna** to niekompletna definicja klasy, zawierająca deklaracje wszystkich metod, ale niezawierająca wszystkich definicji.

Zatem interfejs definiuje **interfejs programistyczny** (ang. *application programming interface* — API) niezależny od jakiejkolwiek hierarchii klas. Interfejsy są najważniejsze w językach programowania obsługujących tylko pojedyncze dziedziczenie, czyli takich, w których klasa może bezpośrednio dziedziczyć tylko po jednej innej klasie. Mówi się, że klasa zawierająca wszystkie metody opisane w interfejsie *implementuje* ten interfejs.

Klasa abstrakcyjna, w odróżnieniu od interfejsu, jest właściwą klasą, tzn. może zawierać dane składowe i definicje metod oraz może być podklasą. Jednak, w odróżnieniu od klasy konkretnej (nieabstrakcyjnej), klasa abstrakcyjna nie zawiera definicji niektórych metod, które są pozostawione do zaimplementowania w jej podklasach. W związku z tym nie można utworzyć egzemplarza klasy abstrakcyjnej — można tylko budować egzemplarze jej konkretnych podklas.

Interfejs jest równoznaczny z klasą abstrakcyjną bez danych składowych i definicji metod. W języku C++ tak właśnie definiuje się interfejsy: deklaruje się klasę bez danych składowych i zawierającą tylko funkcje czysto wirtualne. Oto przykład.

```
class StatusCallback {
  public:
    virtual void updateStatus( int oState, int nState ) = 0;
}
```

Klasa implementuje taki interfejs przez dziedziczenie po nim i dostarczenie definicji zadeklarowanej w nim metody.

```
class MyClass : SomeOtherClass, StatusCallback {
  public:
    void updateStatus( int oState, int nState ){
        if( nState > oState ){
            ..... // Robi coś.
        }
    }
    .... // Reszta klasy.
}
```

W Javie interfejsy definiuje się przy użyciu słowa kluczowego `interface`.

```
public interface StatusCallback {
    void updateStatus( int oState, int nState );
}
```

Taki interfejs można następnie zaimplementować w klasie.

```
public class MyClass implements StatusCallback {
    public void updateStatus( int oState, int nState ){
        if( nState > oState ){
            ..... // Robi coś.
        }
    }
    .... // Reszta klasy.
}
```

W językach programowania obsługujących zarówno interfejsy, jak i klasy abstrakcyjne często tworzy się *domyślne implementacje* interfejsów w klasach abstrakcyjnych. Przykładowo poniższy interfejs:

```
public interface XMLReader {
    public XMLObject fromString( String str );
    public XMLObject fromReader( Reader in );
}
```

może mieć następującą domyślną implementację:

```
public abstract class XMLReaderImpl implements XMLReader {
    public XMLObject fromString( String str ){
        return fromReader( new StringReader( str ) );
    }
    public abstract XMLObject fromReader( Reader in );
}
```

Programista, który chce zaimplementować interfejs `XMLReader`, mógłby utworzyć podklasę klasy `XMLReaderImpl` (prawdopodobnie jako klasę zagnieżdżoną) i zaimplementować tylko jedną metodę zamiast dwóch.

Ogólnie rzecz biorąc, klasy abstrakcyjne są przydatne, gdy wyprowadzane z nich podklasy są bardziej wyspecjalizowanymi typami klasy bazowej (pozostają w relacji *jest*), zwłaszcza gdy w klasie abstrakcyjnej istnieje jakaś wspólna funkcjonalność (np. dane składowe albo definicje metod), z której mogą korzystać klasy pochodne. Interfejsy są przydatne wtedy, kiedy trzeba zapewnić, aby niepowiązane klasy umożliwiały wywoływanie koncepcyjnie powiązanych funkcji w taki sam sposób, ale implementacja tej funkcjonalności może znacznie się różnić między tymi klasami.

## Metody wirtualne

**PROBLEM** Czym są metody wirtualne? Do czego służą?

W programowaniu obiektowym klasy potomne mogą przesłaniać (przedefiniowywać) metody zdefiniowane w klasach nadrzędnych. Jeśli metoda jest wirtualna, definicja metody do wywołania jest określana w czasie działania programu na podstawie typu (klasy) obiektu, na którym została wywołana. W Javie wirtualne są wszystkie metody niestatyczne, ale w językach C# i C++ deklaruje się je przy użyciu słowa kluczowego `virtual` — domyślnie tworzone są metody niewirtualne. Jeżeli metoda nie jest wirtualna, definicja do wywołania jest wybierana w czasie kompilacji na podstawie typu referencji (albo wskaźnika).

Łatwiej to zrozumieć na konkretnych przykładach. Spójrz na poniższe trzy klasy w języku C++.

```
class A {
  public:
    void print() { cout << "A"; }
}
class B : A {
  public:
    void print() { cout << "B"; }
}
class C : B {
  public:
    void print() { cout << "C"; }
}
```

Ponieważ funkcja `print` nie jest wirtualna, to, która wersja zostanie wywołana, zależy od typu użytego w *czasie kompilacji*.

```
A *a = new A();
B *b = new B();
C *c = new C();
a->print(); // "A"
b->print(); // "B"
c->print(); // "C"
((B *)c)->print(); // "B"
((A *)c)->print(); // "A"
((A *)b)->print(); // "A"
```

Gdyby funkcja `print` była wirtualna:

```
class A {
  public:
    virtual void print() { cout << "A"; }
}
class B : A {
  public:
    virtual void print() { cout << "B"; }
}
class C : B {
  public:
    virtual void print() { cout << "C"; }
}
```

to wybór definicji do wywołania zostałby podjęty na podstawie *typu użytego w czasie wykonywania*:

```
A *a = new A();
B *b = new B();
C *c = new C();
```

```
a->print(); // "A"
b->print(); // "B"
c->print(); // "C"
((B *)c)->print(); // "C"
((A *)c)->print(); // "C"
((A *)b)->print(); // "B"
```

Metody wirtualne są mechanizmem wykorzystywanym w polimorfizmie: umożliwiają w jednym wywołaniu metody wywoływanie różnych definicji metod w zależności od tego, do jakiej klasy należy obiekt. W wersji w języku C++ klasy `Shape` przedstawionej na początku tego rozdziału metoda `draw` musiałaby być zdefiniowana jako wirtualna, aby metoda `paintShapes` — która uzyskuje dostęp do obiektów jako referencji typu `Shape` — mogła działać. Specjalnym rodzajem metody wirtualnej w języku C++ jest tzw. funkcja czysto wirtualna, czyli funkcja zadeklarowana, ale celowo niezdefiniowana. (Wprawdzie w języku C++ istnieje możliwość zadeklarowania w klasie funkcji czysto wirtualnej i jednocześnie jej zdefiniowania, ale definicję tę można wywołać tylko w podklasie. Jeśli potrzebujesz czegoś skomplikowanego, język C++ jest w sam raz dla Ciebie). Każda klasa zawierająca funkcję czysto wirtualną lub dziedzicząca taką funkcję i jej niedefiniująca jest klasą abstrakcyjną. (W Javie i C# odpowiednikiem funkcji czysto wirtualnych są metody abstrakcyjne).

Funkcje wirtualne nie są darmowe. Wywołanie takiej funkcji prawie zawsze trwa dłużej niż normalnej, ponieważ trzeba znaleźć adres odpowiedniej funkcji w tablicy. Ponadto tablica ta również zajmuje niewielką ilość pamięci. Mimo to, w większości aplikacji narzut funkcji wirtualnych jest tak mały, że nieistotny.

## Wielodziedziczenie

> **PROBLEM** Dlaczego w językach C# i Java zabronione jest bezpośrednie dziedziczenie po wielu klasach?

W języku C++ klasa może dziedziczyć (pośrednio i bezpośrednio) po więcej niż jednej klasie i nazywa się to **wielodziedziczeniem** (ang. *multiple inheritance*). Natomiast w językach C# i Java klasa może bezpośrednio dziedziczyć tylko po jednej klasie i nazywa się to **dziedziczeniem pojedynczym** (ang. *single inheritance*).

Wielodziedziczenie to technika umożliwiająca tworzenie klas zawierających cechy dwóch różnych hierarchii klas, co często jest przydatne, gdy używa się różnych systemów klas w jednej aplikacji. Jeśli w dwóch takich systemach jest np. zdefiniowana inna klasa bazowa dla wyjątków, to za pomocą wielodziedziczenia można utworzyć klasy wyjątków nadające się do użytku w obu tych systemach.

Problem z wielodziedziczeniem polega na tym, że może prowadzić do niejednoznaczności. Klasycznym przykładem jest sytuacja, w której klasa dziedziczy po dwóch innych klasach, z których każda dziedziczy po tej samej klasie.

```
class A {
  protected:
    bool flag;
};
class B : public A {};
class C : public A {};
class D : public B, public C {
```

```
  public:
    void setFlag( bool nflag ){
      flag = nflag; // Niejednoznaczne.
    }
};
```

W tym przykładzie składowa `flag` jest zdefiniowana w klasie `A`. Klasa `D` dziedziczy po klasach `B` i `C`, które obie dziedziczą po klasie `A`, więc zasadniczo w klasie `D` znajdują się dwie kopie składowej `flag`, ponieważ w hierarchii klasy `D` klasa `A` występuje dwa razy. Którą z tych składowych należy ustawiać? Kompilator zgłosi błąd polegający na tym, że odniesienie do składowej `flag` w klasie `D` jest niejednoznaczne. Jednym z rozwiązań jest własnoręczne rozstrzygnięcie wątpliwości:

```
B::flag = nflag;
```

Inną możliwością jest zadeklarowanie `B` i `C` jako *wirtualnych klas bazowych*, dzięki czemu w hierarchii będzie mogła występować tylko jedna kopia `A`, co również wyeliminuje niejednoznaczność.

Istnieją jeszcze inne problemy z wielodziedziczeniem, np. dotyczące kolejności inicjacji klas bazowych podczas tworzenia obiektu klasy pochodnej czy niezamierzonego ukrywania składowych przed klasą pochodną. Aby uniknąć tego wszystkiego, w niektórych językach programowania zezwolono tylko na pojedyncze dziedziczenie. Podczas gdy to znacznie upraszcza kwestie dziedziczenia, jednocześnie je ogranicza, ponieważ tylko klasy o wspólnym przodku mogą mieć wspólne zachowania. Trochę łagodzą to interfejsy, które umożliwiają klasom z różnych hierarchii udostępnianie wspólnych interfejsów nawet wtedy, gdy nie są zaimplementowane przy użyciu wspólnego kodu.

# PODSUMOWANIE

Obiektowe języki programowania są powszechnie używane, więc aby dostać pracę, zazwyczaj należy dobrze znać podstawy obiektowości. Kandydat musi wiedzieć, czym różni się klasa od obiektu, oraz znać pojęcia polimorfizmu i dziedziczenia.

Należy też znać różnice między językami programowania w odniesieniu do różnych aspektów obiektowości.

# Skorowidz

## D



## E



## F



## G



## H



## I



## J



## K

## L



## Ł



## M

## Q



## R



## S

## Ś



## T



## U



## W



## Z

## Lektura obowiązkowa przed rozmową rekrutacyjną!

Rozmowa rekrutacyjna to ten rodzaj spotkania, które wywołuje stres nawet u najbardziej odpornych. Podchwytliwe pytania, zadania do rozwiązania, skomplikowane testy dotyczące spraw technicznych oraz nierzadko kilkugodzinny dialog wymagają dużej odporności i wiedzy. Zastanawiasz się, jak przygotować się do tej rozmowy? Na to pytanie nie ma jednej odpowiedzi, ale wszystkie znajdziesz w tej książce!

Ta lektura doskonale przygotuje Cię do kolejnej rozmowy rekrutacyjnej. Na samym początku dowiesz się, jak szukać ofert pracy oraz jak rozpocząć proces ubiegania się o wybrane stanowisko. W kolejnych rozdziałach zaznajomisz się z różnymi problemami programistycznymi oraz sposobami ich rozwiązania. Zobacz, czym mogą próbować Cię zaskoczyć rekruterzy w obszarach związanych z listami, drzewami, grafami i rekurencją. Poznaj najczęściej zadawane pytania dotyczące wzorców projektowych, programowania obiektowego oraz baz danych. Książka ta jest obowiązkową lekturą dla wszystkich osób chcących pomyślnie przejść proces rekrutacji na wymarzone stanowisko!

Dzięki tej książce:

- nauczysz się skutecznie szukać odpowiednich dla Ciebie ofert
- stworzysz profesjonalny życiorys
- poznasz pytania najczęściej pojawiające się podczas rozmów rekrutacyjnych
- zrobisz milowy krok do przodu w kierunku wymarzonej pracy



Helion
27246 numer katalogowy
księgarnia internetowa
http://helion.pl
zamówienia telefoniczne
0 801 339900
0 601 339900
Informatyka w najlepszym wydaniu

Sprawdź najnowsze promocje:
http://helion.pl/promocje
Książki najchętniej czytane:
http://helion.pl/bestsellery
Zamów informacje o nowościach:
http://helion.pl/nowosci

Helion SA
ul. Kościuszki 1c, 44-100 Gliwice
tel.: 32 230 98 63
e-mail: helion@helion.pl
http://helion.pl

ISBN 978-83-246-9861-5
9 788324 698615
cena: 49,00 zł

Wrox
An Imprint of
WILEY