Programowanie II

Rafał NOWAK

Plan zajęć :

- Wprowadzenie do środowiska Visual Studio
- Ogólne spojrzenie na programowanie obiektowe
- Klasy i obiekty
- Przeciążanie operatorów
- Dynamiczny przydział pamięci
- Dziedziczenie
- Obsługa błędów – wyjątki
- Programowanie generyczne, szablony
- Biblioteka STL
- Graficzny interfejs użytkownika – Qt

Wprowadzenie do środowiska Visual Studio

Wprowadzenie do środowiska Visual Studio

Plan zajęć :

- Tworzenie projektu
- Kompilacja
- Debugowanie

Wprowadzenie do środowiska Visual Studio

Podsumowanie :

- Tworzenie projektu
- Kompilacja
- Debugowanie

Ogólne spojrzenie na programowanie obiektowe

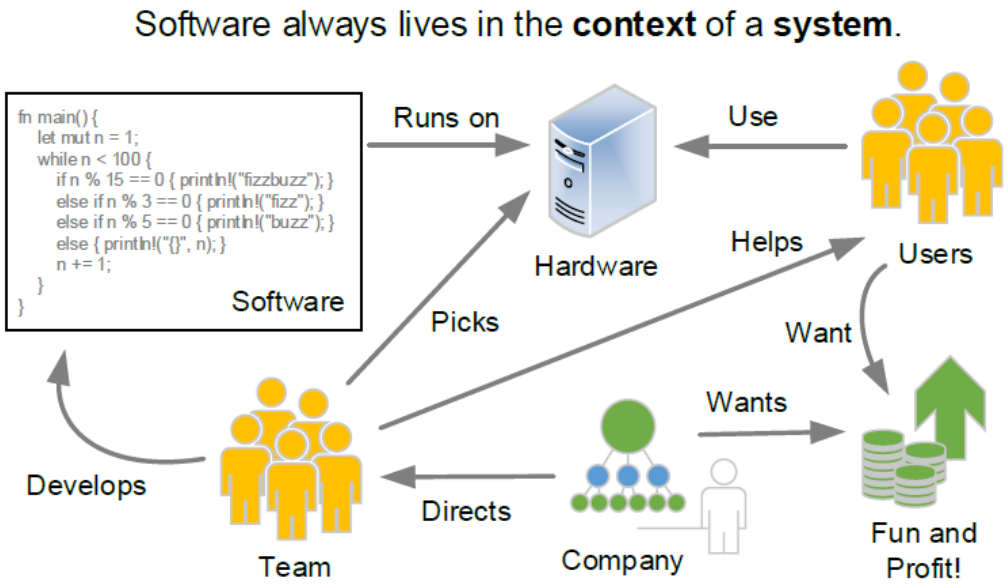
Ogólne spojrzenie na programowanie obiektowe

Plan zajęć :

- Po co tworzymy oprogramowanie?
- Cechy dobrego oprogramowania
- Programowanie proceduralne
- Tworzenie modułów
- Programowanie obiektowe
- UML

Po co tworzymy oprogramowanie?

Po co tworzymy programowanie?



źródło: „Design It! From Programmer to Software Architect”, Michael Keeling

Cechy dobrego oprogramowania

Możliwość ponownego użycia

- Zbudowane jest z przenośnych i niezależnych komponentów, które mogą być ponownie wykorzystane w innych programach

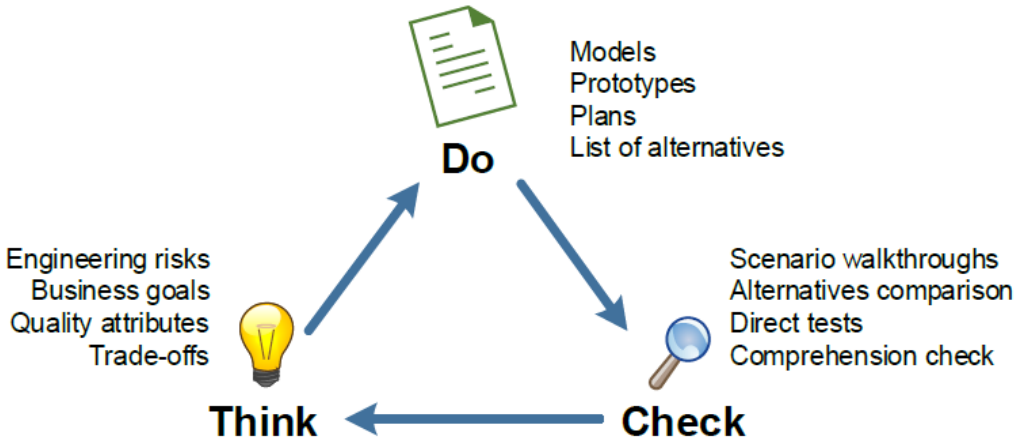
Rozszerzalność

- Udostępnia możliwość wykorzystania zewnętrznych modułów rozszerzających (np. dodatki i skrypty Microsoft Office)

Elastyczność

- Łatwość wprowadzania zmian przy dodawaniu nowych funkcjonalności
- Lokalne efekty wprowadzanych modyfikacji
- Małe prawdopodobieństwo uszkodzenia całego systemu przy zmianach w projekcie

Jak tworzyć (dobre) oprogramowanie?



źródło: „Design It! From Programmer to Software Architect”, Michael Keeling

Proces projektowania

Cel: zbudować system

Proces projektowania przebiega następująco:

- Podział/opis systemu jako zespołu komponentów
- Podział/opis komponentów jako zespołu mniejszych komponentów

Pojęcie abstrakcji

- Podstawowe w procesie projektowania, ukrywa szczegóły komponentów nieistotne w bieżącej fazie projektowania

Identyfikacja komponentów metodą zstępującą

- Stopniowy podział systemu na coraz mniejsze, prostsze komponenty

Integracja komponentów metodą wstępującą

- Budowa systemu przez składanie komponentów na różne sposoby

Projekt odbywa się zgodnie z paradygmatem: proceduralnym, obiektowym

źródło: „Programowanie obiektowe”, Grzegorz Jabłoński

Abstrakcja

Nazwany zbiór atrybutów i sposobu zachowania konieczny do modelowania obiektu w określonym celu

Pożądane właściwości:

- Dobrze nazwany — nazwa oddaje cechy abstrakcji
- Spójny — sensowny
- Dokładny — zawiera tylko atrybuty modelowanego obiektu
- Minimalny — zawiera tylko atrybuty niezbędne dla określonego celu
- Kompletny — zawiera wszystkie atrybuty niezbędne dla określonego celu

źródło: „Programowanie obiektowe”, Grzegorz Jabłoński

Formy abstrakcji

Funkcje

- Zdefiniowanie zbioru funkcji w celu wykonania zadania
- Przekazywanie informacji między funkcjami
- Wynik: hierarchiczna organizacja funkcji

Moduły

- Zdefiniowanie modułów, w których są dane i procedury
- Każdy moduł posiada sekcję prywatną i publiczną
- Moduł grupuje powiązane dane i procedury
- Moduł działa jako mechanizm zasięgu

Klasy/obiekty

- Abstrakcyjne typy danych
- Podział projektu na zbiór współpracujących klas
- Każda klasa pełni bardzo szczególne funkcje
- Klasy mogą być użyte do tworzenia wielu egzemplarzy obiektów

źródło: „Programowanie obiektowe”, Grzegorz Jabłoński

Problemy podejścia proceduralnego

- Otrzymujemy duży program złożony z wielu małych procedur
- Brak naturalnej hierarchii organizującej te procedury
- Często nie jest jasne, która procedura co wykonuje na których danych
- Słaba kontrola potencjalnego dostępu procedur do danych
- Powyższe cechy powodują, że usuwanie błędów, modyfikacja i pielęgnacja są trudne
- Naturalna wzajemna zależność procedur spowodowana przekazywaniem danych (albo co gorsza danymi globalnymi) powoduje, że trudno jest je ponownie użyć w innych systemach

źródło: „Programowanie obiektowe”, Grzegorz Jabłoński

Programowanie modularne

- Względnie proste rozszerzenie czystego podejścia proceduralnego
- Dane i zebrane z nimi procedury są zebrane w modułach
- Moduł zapewnia jakąś metodę ukrycia jego zawartości
- W szczególności, dane mogą być modyfikowane tylko przez procedury w tym samym module

Problemy w projektowaniu modularnym

- Moduły rozwiązują większość problemów z programowaniem proceduralnym wymienionych uprzednio
- Moduły pozwalają na jedynie częściowe ukrywanie informacji w porównaniu z podejściem obiektowym
- Nie można mieć kilku kopii jednego modułu, co ogranicza projektanta

źródło: „Programowanie obiektowe”, Grzegorz Jabłoński

Paradygmat obiektowy

- Analogia do konstruowania maszyny z części składowych
- Każda część jest obiektem, który posiada swoje atrybuty i właściwości oraz współdziała z innymi częściami w celu rozwiązania problemu
- Identyfikacja klas obiektów, które mogą być ponownie użyte
- Myślenie używające pojęć obiektów i ich wzajemnego oddziaływania
- Na wysokim poziomie abstrakcji, myślenie o obiektach jako bytach samych w sobie, nie wewnętrznych strukturach potrzebnych do działania obiektu

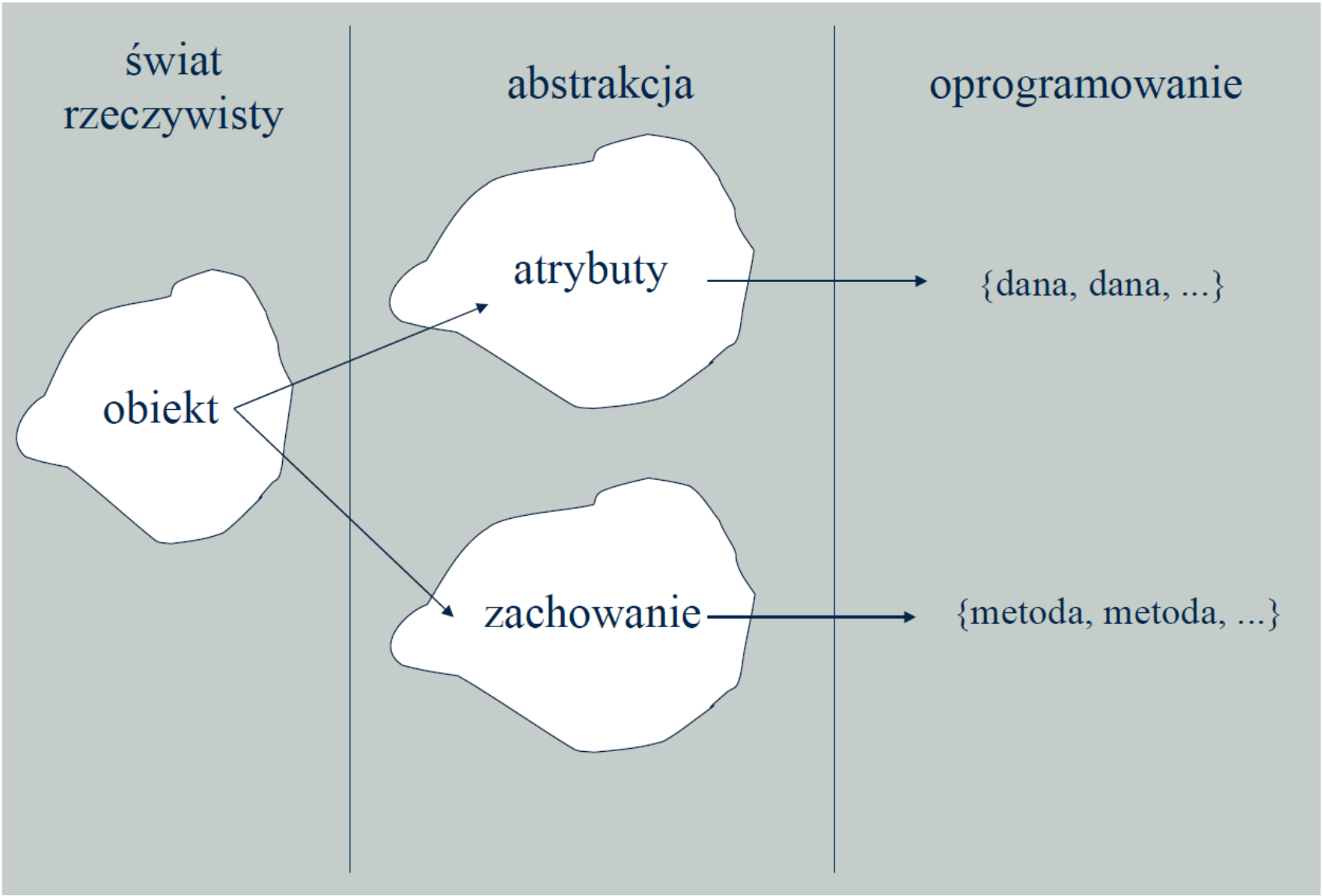
źródło: „Programowanie obiektowe”, Grzegorz Jabłoński

Dlaczego podejście obiektowe?

- To po prostu kolejny paradygmat … (i zapewne będą kolejne)
- Każdy system zaprojektowany i zaimplementowany obiektowo może być zbudowany używając czystego podejścia proceduralnego
- Podejście obiektowe jednak ułatwia pewne rzeczy
- Podczas projektowania na wysokim poziomie, często bardziej naturalne jest myślenie o problemie używając pojęć zespołu oddziałujących na siebie rzeczy (obiektów), niż pojęć danych i procedur
- Podejście obiektowe często ułatwia zrozumienie i kontrolę nad dostępem dodanych
- Podejście obiektowe promuje ponowne użycie

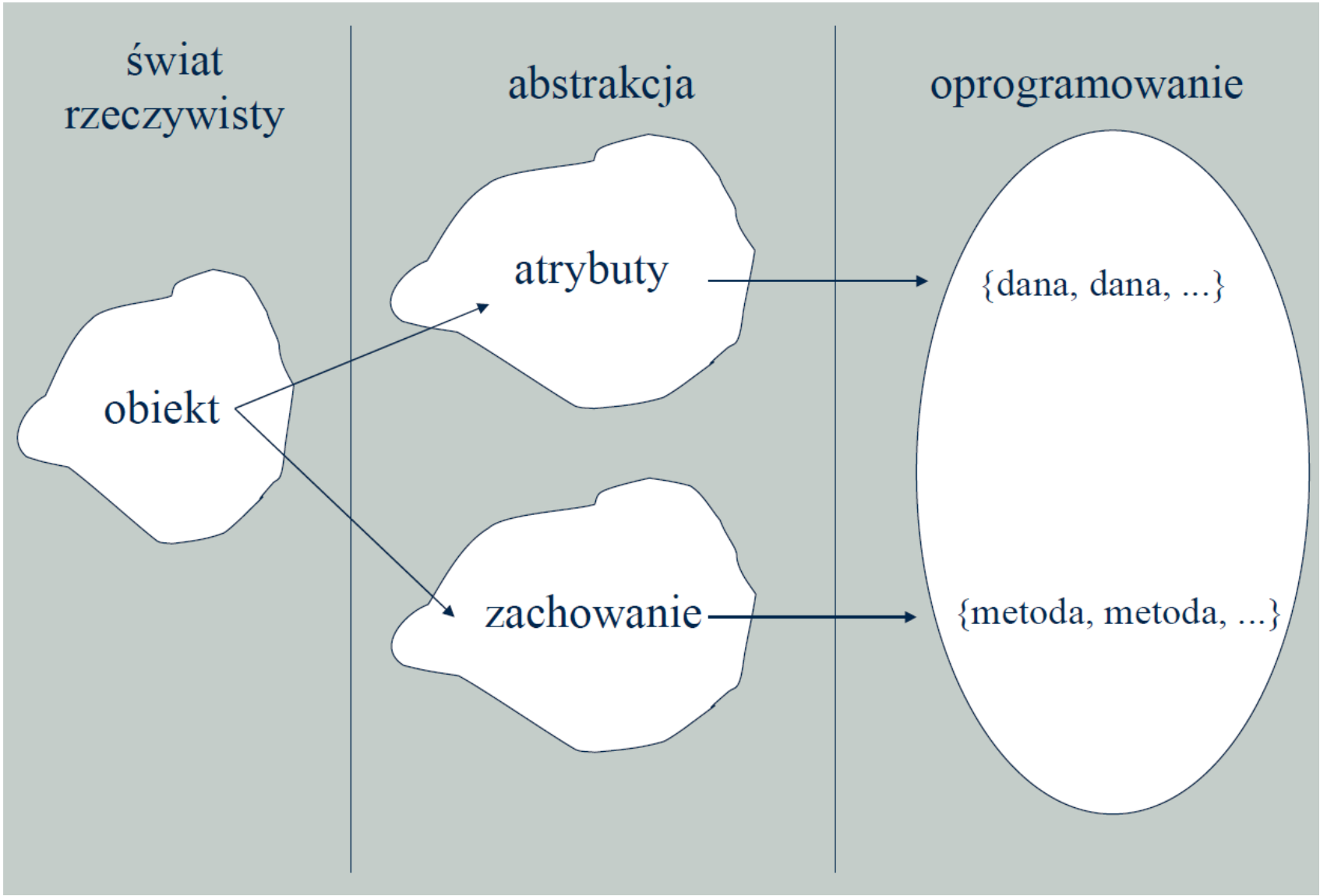
źródło: „Programowanie obiektowe”, Grzegorz Jabłoński

Odwzorowanie abstrakcji i oprogramowania



źródło: „Programowanie obiektowe”, Grzegorz Jabłoński

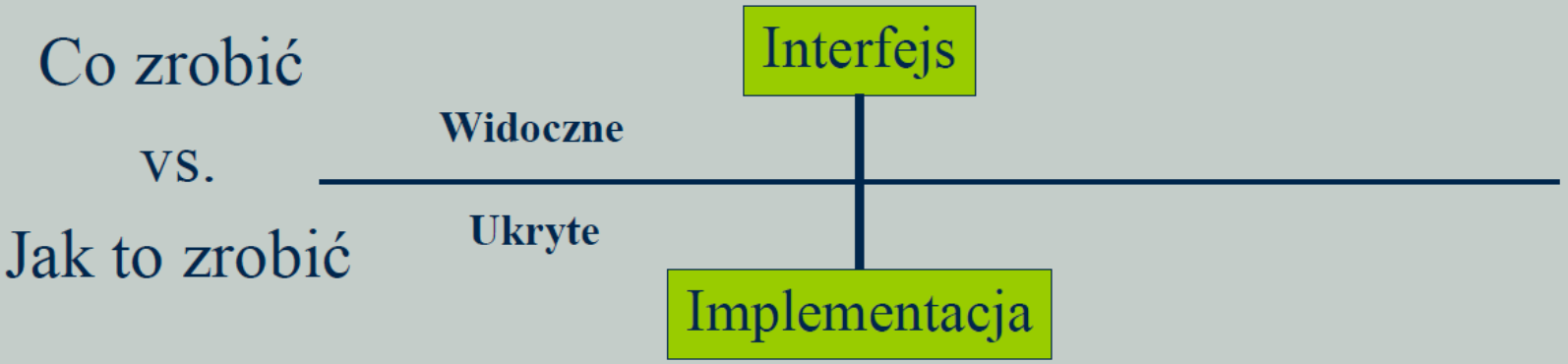
Odwzorowanie abstrakcji i oprogramowania (OO)



źródło: „Programowanie obiektowe”, Grzegorz Jabłoński

Oddzielenie interfejsu od implementacji

- W programowaniu: niezależna specyfikacja interfejsu i jednej lub wielu implementacji tego interfejsu.



- Dodatkową korzyścią jest możliwość programowania uzależnionego od interfejsu bez zajmowania się jego implementacją
  - Programowanie oparte na kontrakcie
  - Umożliwia abstrakcję w procesie projektowania

źródło: „Programowanie obiektowe”, Grzegorz Jabłoński

Ujednolicony Język Modelowania – UML

(ang. Unified Modeling Language)

- Język pół-formalny wykorzystywany do modelowania różnego rodzaju systemów
- Służy do modelowania dziedziny problemu (opisywania-modelowania fragmentu istniejącej rzeczywistości – na przykład modelowanie tego, czym zajmuje się jakiś dział w firmie) – w przypadku stosowania go do analizy oraz do modelowania rzeczywistości, która ma dopiero powstać – tworzy się w nim głównie modele systemów informatycznych.
- UML jest przeważnie używany wraz ze swoją reprezentacją graficzną – jego elementom przypisane są odpowiednie symbole wiązane ze sobą na diagramach.
- UML nie jest metodą samą w sobie, lecz był projektowany dla kompatybilności z wiodącymi obiektowymi metodami rozwoju oprogramowania

źródło: WIKIPEDIA

Diagramy

Diagramy struktur

- Klas (najczęściej spotykane, ang. class diagram)
- Obiektów (ang. object diagram)
- Komponentów (ang. component diagram)
- Wdrożenia (ang. deployment diagram)
- Struktur złożonych (ang. composite structure diagram)
- Pakietów (ang. package diagram)
- Profili (ang. profile diagram, nowość wprowadzona w UML 2.2)

źródło: WIKIPEDIA

Diagramy

Diagramy zachowań

- Czynności (ang. activity diagram)
- Przypadków użycia (ang. use case diagram)
- Maszyny stanów (ang. state machine diagram)
- Interakcji (diagram abstrakcyjny)
- Komunikacji (ang. communication diagram)
- Sekwencji (ang. sequence diagram)
- Czasowe (ang. timing diagram)
- Przeglądu interakcji (ang. interaction overview diagram)

źródło: WIKIPEDIA

Diagramy

Użycie

W praktyce rzadko kiedy trzeba opracowywać wszystkie diagramy i w większości przypadków korzysta się z mniej niż połowy wyżej wymienionych. Nie powinno modelować się tylko dla samego modelowania, dlatego nie zawsze wszystkie rodzaje są potrzebne.

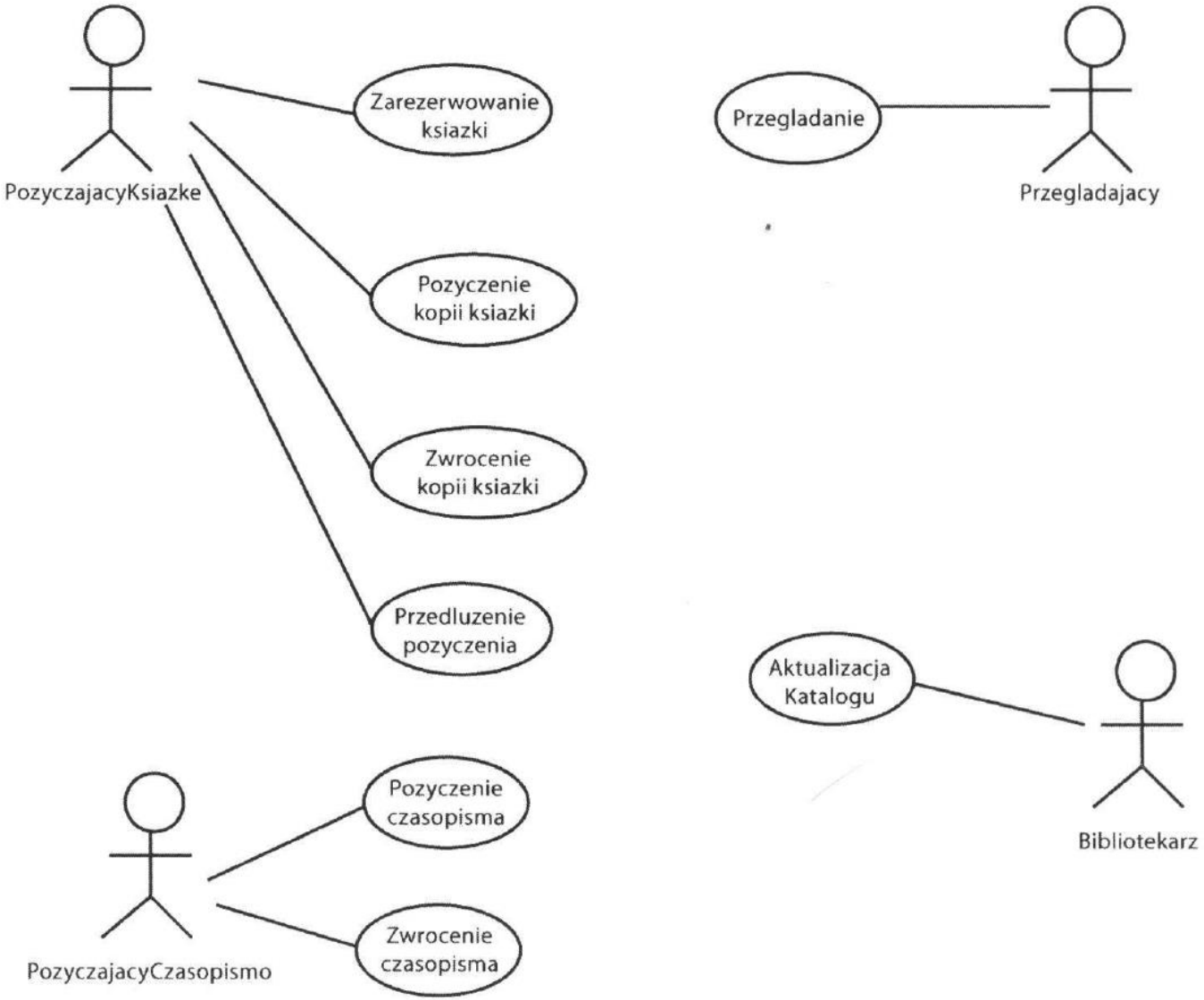
Projektując system informatyczny, rozpoczyna się przeważnie od tworzenia diagramów w następującej kolejności:

1. Przypadków użycia
2. Sekwencji
3. Klas
4. Czynności

Są to najczęściej wykorzystywane diagramy. Pozostałe bywają pomijane, zwłaszcza przy budowaniu niedużych systemów informatycznych.

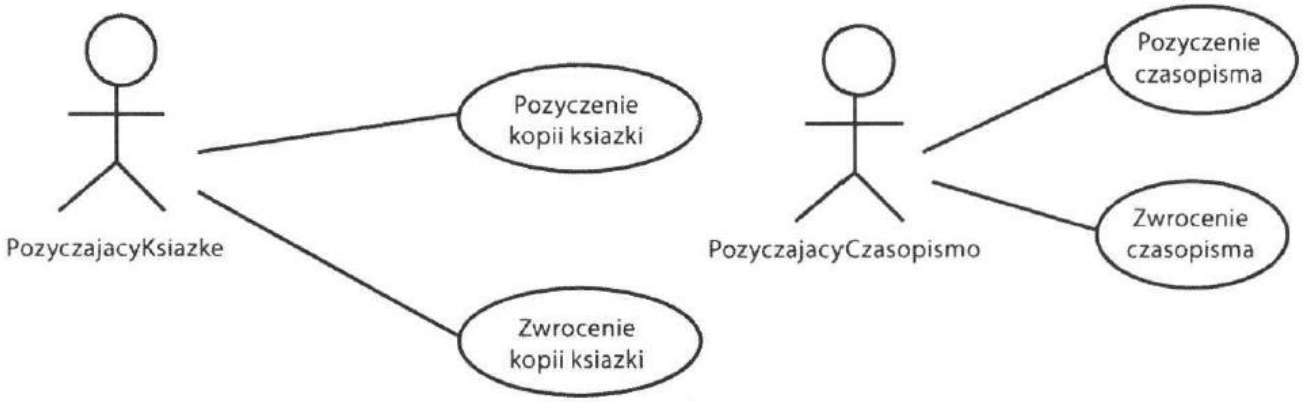
źródło: WIKIPEDIA

Przykład – diagram przypadków użycia dla biblioteki



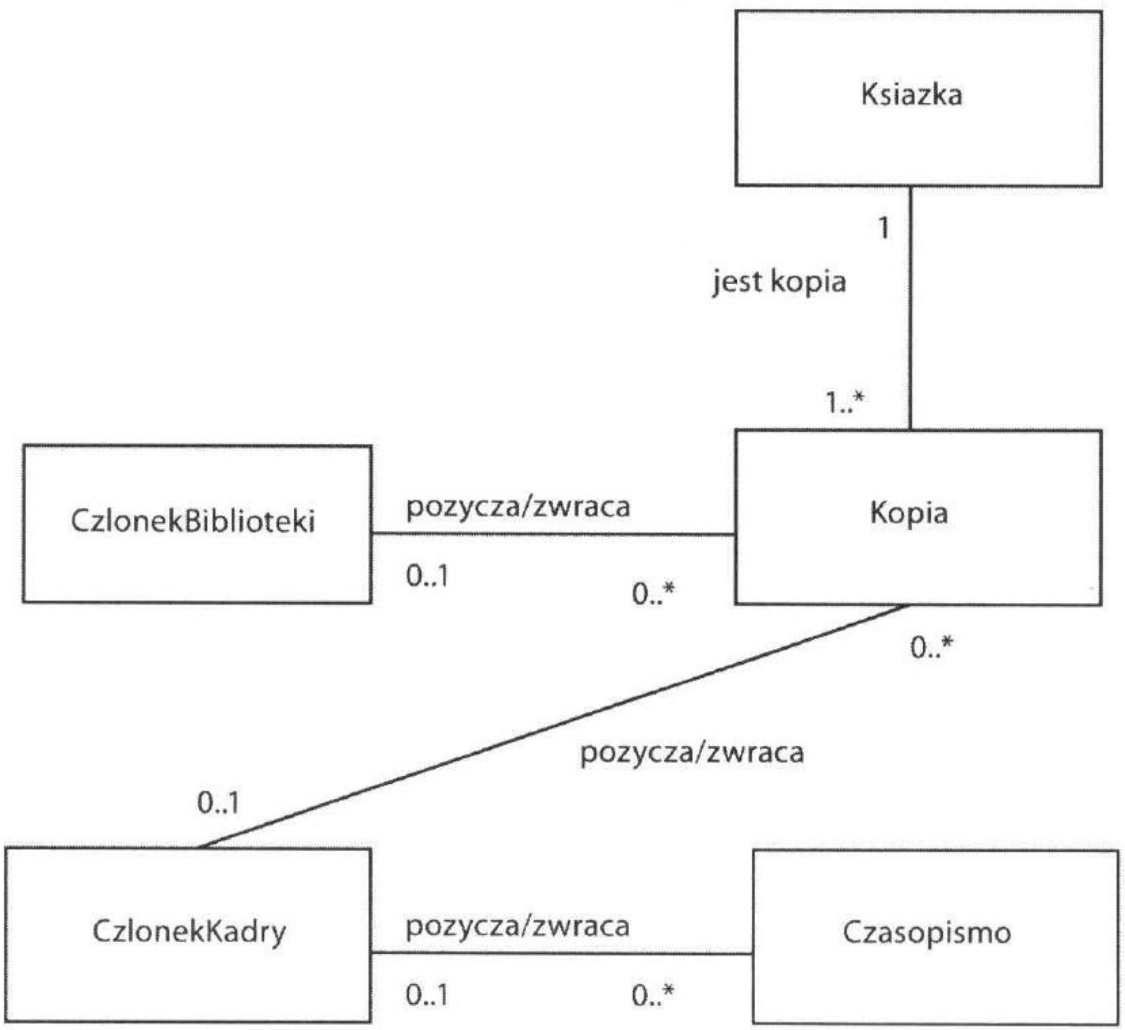
źródło: „UML Inżynieria oprogramowania” Perdita Stevens

Przykład – diagram przypadków użycia dla pierwszej iteracji



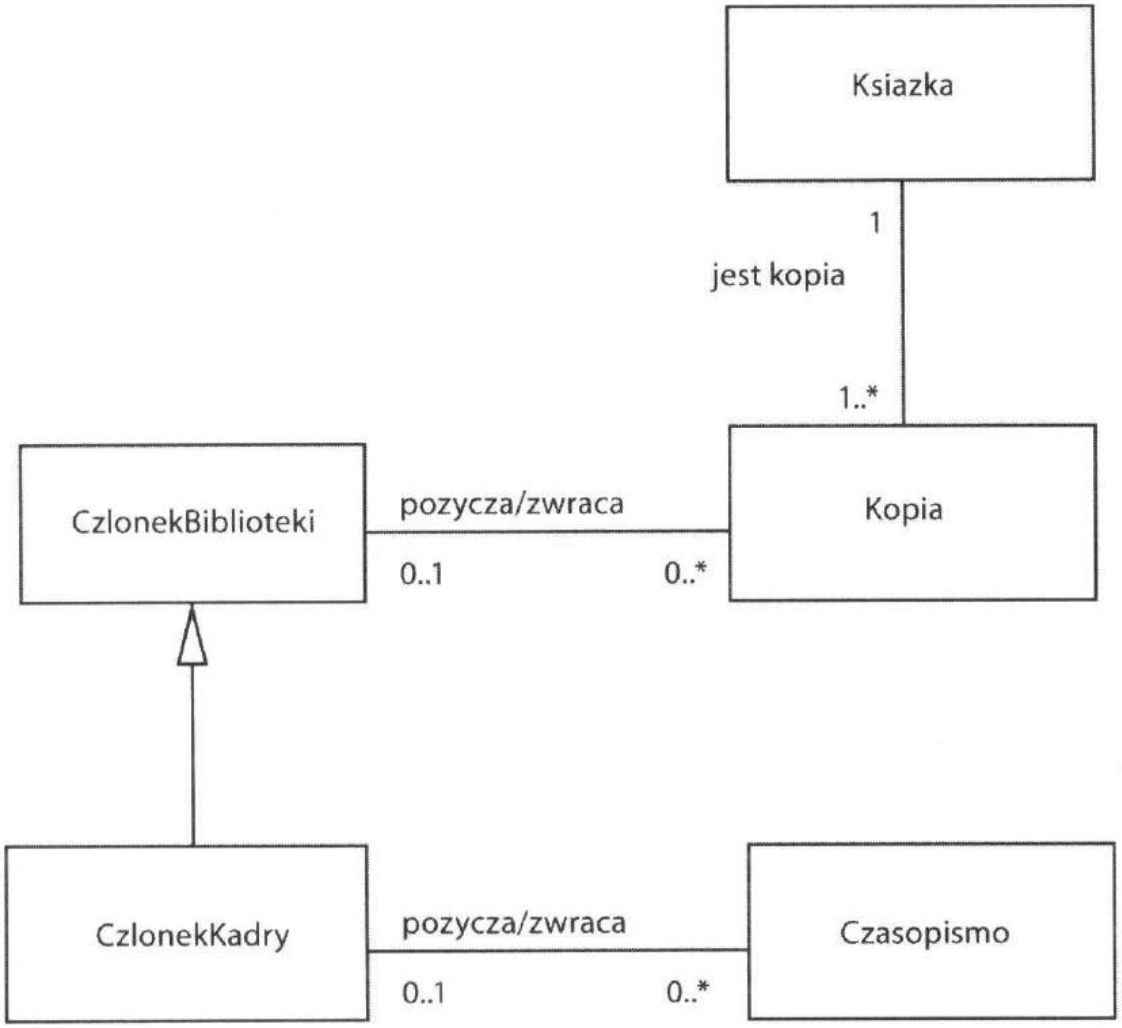
źródło: „UML Inżynieria oprogramowania” Perdita Stevens

Przykład – wstępny model klas biblioteki



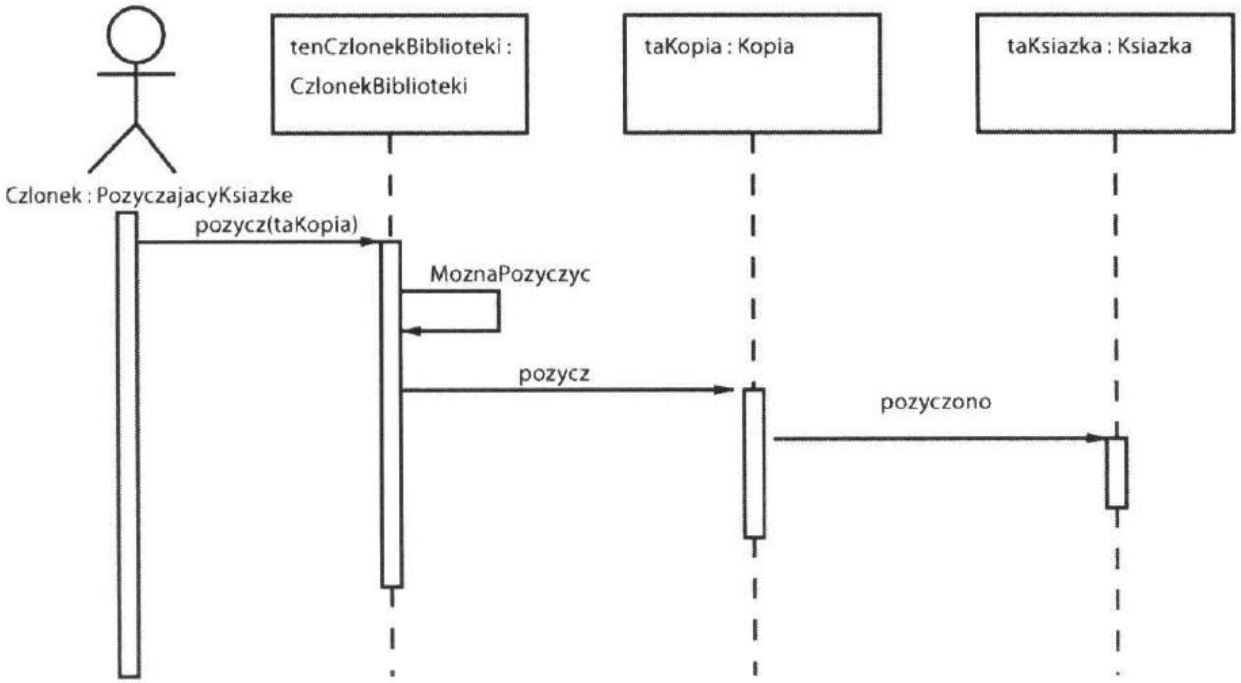
źródło: „UML Inżynieria oprogramowania” Perdita Stevens

Przykład – poprawiony model klas biblioteki



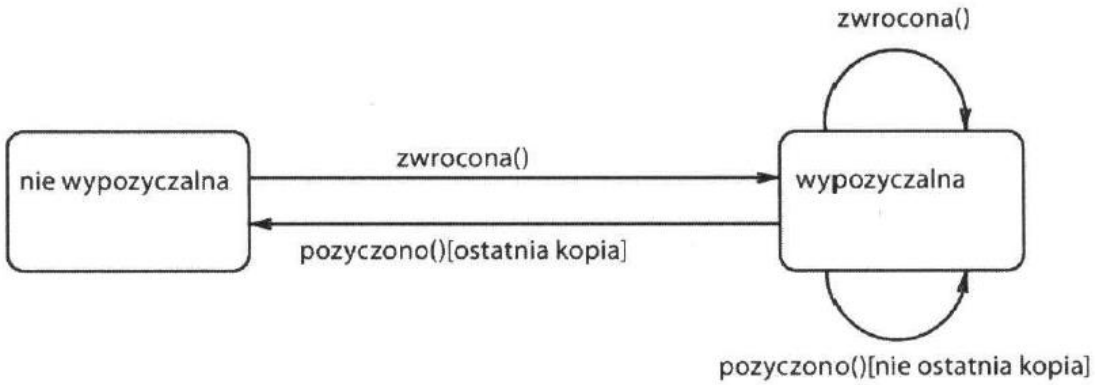
źródło: „UML Inżynieria oprogramowania” Perdita Stevens

Przykład – interakcje pokazane na diagramie sekwencji



źródło: „UML Inżynieria oprogramowania” Perdita Stevens

Przykład – diagram stanu klasy TypKsiazka



źródło: „UML Inżynieria oprogramowania” Perdita Stevens

Ogólne spojrzenie na programowanie obiektowe

Podsumowanie :

- Po co tworzymy oprogramowanie?
- Cechy dobrego oprogramowania
- Programowanie proceduralne
- Tworzenie modułów
- Programowanie obiektowe
- UML

Klasy i obiekty

Klasy i obiekty

Plan zajęć :

- Składowe klas
- Konstruktory i destruktor
- Metody z modyfikatorem **const**
- Tożsamość obiektu – wskaźnik **this**
- Tworzenie tablic obiektów
- Zasięg Klasy

Klasy i obiekty

Klasy to najważniejsze pojedyncze rozszerzenie języka C++ w stosunku do C , implementujące i scalające wszystkie własności programowania obiektowego.

Najważniejszymi cechami programowania obiektowego są:

- Abstrakcja
- Hermetyzacja i ukrywanie danych
- Polimorfizm
- Dziedziczenie
- Zdatność kodu do wielokrotnego wykorzystania

Klasy i obiekty

Abstrakcja polega na wyodrębnieniu i uogólnieniu najważniejszych cech problemu i wyrażeniu rozwiązania właśnie w obrębie tych cech.

Typ

W skrócie, podanie typu określa trzy elementy:

- Ilość pamięci niezbędnej do reprezentacji obiektu danego typu w pamięci
- Sposób interpretacji bitów pamięci zajmowanej przez obiekt danego typu
- Zakres operacji (metod), którym może podlegać obiekt danych

Klasa jest w języku C++ środkiem translacji abstrakcji na postać typu definiowanego przez użytkownika. Klasa łączy w jednej jednostce reprezentację danych i zbiór operacji na tych danych.

Klasa Point – podejście pierwsze

Klasa reprezentująca punkt w kartezjańskim układzie współrzędnych.

Ma posiadać pola określające:

- Nazwę punktu
- Współrzędną x punktu
- Współrzędną y punktu

Ma posiadać metody umożliwiające:

- Nadawanie nazwy
- Ustalanie współrzędnych
- Pobieranie współrzędnych
- Wyświetlanie informacji o punkcie

Klasa Point (plik: Point.h)

```
#pragma once
#include <string>

class Point
{
  private:
    std::string m_name;
    double m_x;
    double m_y;
  public:
    void setName(const std::string& name);
    void setXY(double x, double y);
    double getX();
    double getY();
    void show();
};
```

Klasa Point (plik: Point.cpp)

```cpp
#include <iostream>
#include "Point.h"

void Point::setName(const std::string& name) {
  m_name = name;
}

void Point::setXY(double x, double y) {
  m_x = x;
  m_y = y;
}

double Point::getX() {
  return m_x;
}

double Point::getY() {
  return m_y;
}

void Point::show() {
  std::cout << "Point: " << m_name << "(" << m_x << ", " << m_y << ")" << std::endl;
}
```

Klasa Point – podejście drugie

Klasa reprezentująca punkt w kartezjańskim układzie współrzędnych.

Ma posiadać pola określające:

- Nazwę punktu
- Współrzędną x punktu
- Współrzędną y punktu

Ma posiadać metody umożliwiające:

- Nadawanie nazwy
- Ustalanie współrzędnych
- Pobieranie współrzędnych
- Wyświetlanie informacji o punkcie

Klasa ma gwarantować upodobnienie tworzonych obiektów do wartości typów wbudowanych.

Klasa Point (plik: Point.h)

```
#pragma once
#include <string>

class Point
{
private:
  std::string m_name;
  double m_x;
  double m_y;
public:
  Point();
  Point(const std::string& name, double x = 0, double y = 0);
  ~Point();
  void setName(const std::string& name);
  void setXY(double x, double y);
  double getX();
  double getY();
  void show();
};
```

Klasa Point (plik: Point.cpp)

```
#include <iostream>
#include "Point.h"

Point::Point() {
  std::cout << "Wywolano konstruktor domyslny\n";
  m_name = "bez nazwy";
  m_x = 0;
  m_y = 0;
}

Point::Point(const std::string& name, double x, double y) {
  std::cout << "Wywolano konstruktor z argumentem " << name << std::endl;
  m_name = name;
  m_x = x;
  m_y = y;
}

Point::~Point() {
  std::cout << "Likwiduje " << m_name << std::endl;
}

…
```

Klasa Point (plik: Point.cpp)

```
...

void Point::setName(const std::string& name) {
  m_name = name;
}

void Point::setXY(double x, double y) {
  m_x = x;
  m_y = y;
}

double Point::getX() {
  return m_x;
}

double Point::getY() {
  return m_y;
}

void Point::show() {
  std::cout << "Point: " << m_name << "(" << m_x << ", " << m_y << ")" << std::endl;
}
```

Klasa Point – podejście trzecie

Klasa reprezentująca punkt w kartezjańskim układzie współrzędnych.

- Metody z modyfikatorem **const**
- Tożsamość obiektu – wskaźnik **this**

Klasa Point (plik: Point.h)

```
#pragma once
#include <string>

class Point
{
private:
  std::string m_name;
  double m_x;
  double m_y;
public:
  Point();
  Point(const std::string& name, double x = 0, double y = 0);
  ~Point();
  void setName(const std::string& name);
  void setXY(double x, double y);
  double getX();
  double getY();
  void show() const;
  double distance(const Point& p) const;
  const Point & distant(const Point& p) const;
};
```

Klasa Point (plik: Point.cpp)

```
#include <iostream>
#include "Point.h"

Point::Point() {
  //std::cout << "Wywolano konstruktor domyslny\n";
  m_name = "bez nazwy";
  m_x = 0;
  m_y = 0;
}

Point::Point(const std::string& name, double x, double y) {
  //std::cout << "Wywolano konstruktor z argumentem " << name << std::endl;
  m_name = name;
  m_x = x;
  m_y = y;
}

Point::~Point() {
  //std::cout << "Likwiduje " << m_name << std::endl;
}

void Point::setName(const std::string& name) {
  m_name = name;
}

void Point::setXY(double x, double y) {
  m_x = x;
  m_y = y;
}
```

Klasa Point (plik: Point.cpp)

```
...

double Point::getX() {
  return m_x;
}

double Point::getY() {
  return m_y;
}

void Point::show() const {
  std::cout << "Point: " << m_name << "(" << m_x << ", " << m_y << ")" << std::endl;
}

double Point::distance(const Point& p) const {
  return sqrt( (p.m_x - m_x) * (p.m_x - m_x) + (p.m_y - m_y) * (p.m_y - m_y) );
}

const Point& Point::distant(const Point& p) const {
  double d = sqrt( m_x * m_x + m_y * m_y);
  double dp = sqrt(p.m_x * p.m_x + p.m_y * p.m_y);

  if (d > dp)
    return *this;

  return p;
}
```

Klasa Point – podejście czwarte

Klasa reprezentująca punkt w kartezjańskim układzie współrzędnych.

- Tworzenie tablic obiektów
- Zasięg Klasy

Klasa Point (plik: Point.h)

```
#pragma once
#include <string>

class Point
{
private:
  static int m_numberOfPoints;
  std::string m_name;
  double m_x;
  double m_y;
public:
  static int numberOfPoints();
  Point();
  Point(const std::string& name, double x = 0, double y = 0);
  ~Point();
  void setName(const std::string& name);
  void setXY(double x, double y);
  double getX();
  double getY();
  void show() const;
  double distance(const Point& p) const;
  const Point& distant(const Point& p) const;
};
```

Klasa Point (plik: Point.cpp)

```
#include <iostream>
#include "Point.h"

int Point::m_numberOfPoints = 0;

Point::Point() {
  m_name = "bez nazwy";
  m_x = 0;
  m_y = 0;
  Point::m_numberOfPoints++;
}

Point::Point(const std::string& name, double x, double y) {
  m_name = name;
  m_x = x;
  m_y = y;
  Point::m_numberOfPoints++;
}

Point::~Point() {
  Point::m_numberOfPoints--;
}

…

int Point::numberOfPoints() {
  return Point::m_numberOfPoints;
}
```

Klasa Point (plik: main.cpp)

```
#include <iostream>
#include "Point.h"
using namespace std;

int main()
{
  Point points[10] = {
    Point("P1", 0, 0),
    Point("P2", 10, 20),
    Point("P3", 30, 40)
  };

  cout << "Istnieje " << Point::numberOfPoints() << " punktow."<< endl;

  for (int i = 0; i < 10; i++)
  {
    points[i].show();
  }


  {
    Point p1, p2, p3;
    cout << endl;
    cout << "Teraz istnieje " << Point::numberOfPoints() << " punktow." << endl;
  }

  cout << "A teraz istnieje " << Point::numberOfPoints() << " punktow." << endl;

}
```

Klasy i obiekty

Podsumowanie :

- Składowe klas
- Konstruktory i destruktor
- Metody z modyfikatorem **const**
- Tożsamość obiektu – wskaźnik **this**
- Tworzenie tablic obiektów
- Zasięg Klasy

Przeciążanie operatorów

Przeciążanie operatorów

Plan zajęć :

- Wykorzystanie metod do przeprowadzania operacji arytmetycznych
- Przeciążanie operatorów – implementacja wykorzystująca metody
- Przeciążanie operatorów – implementacja wykorzystująca funkcje
- Przeciążanie operatorów **<<** i **>>**

Klasa Fraction – podejście pierwsze

Klasa reprezentująca liczby wymierne.

Ma posiadać pola określające:

- licznik
- mianownik

Ma posiadać metody umożliwiające:

- dodawanie
- odejmowanie

Stanem wewnętrznym liczby wymiernej jest jej wartość reprezentowana parą liczb całkowitych l,m – licznik i mianownik. Reprezentacja taka nie jest jednak jednoznaczna; postulujemy niezmiennik stanu wewnętrznego: licznik i mianownik nieskracalne, mianownik nieujemny. Zatem licznik i mianownik powinny być prywatne – w przeciwnym wypadku nie można gwarantować spełnienia niezmiennika, gdyż dowolna funkcja byłaby uprawniona do niekontrolowanej modyfikacji tych składowych.

Klasa Fraction – podejście pierwsze

Stanem wewnętrznym liczby wymiernej jest jej wartość reprezentowana parą liczb całkowitych l,m – licznik i mianownik. Reprezentacja taka nie jest jednak jednoznaczna;

postulujemy **niezmiennik stanu wewnętrznego**:

- licznik i mianownik nieskracalne,
- mianownik nieujemny.

Zatem licznik i mianownik powinny być prywatne – w przeciwnym wypadku nie można gwarantować spełnienia niezmiennika, gdyż dowolna funkcja byłaby uprawniona do niekontrolowanej modyfikacji tych składowych.

Klasa Fraction – podejście pierwsze

Pozostaje do rozstrzygnięcia interpretacja 0 w mianowniku.

Matematycznie nie jest to liczba wymierna, ale możemy potraktować kombinacje **1/0**, **-1/0** i **0/0** jako oznaczenia symboliczne odpowiednio **+∞**, **-∞** i **wartości nieokreślonej**.

Jakakolwiek wartość licznika różna od zera będzie redukowana do 1 albo

-1 w zależności od znaku.

Klasa Point (plik: Fraction.h)

```
#pragma once
#include <iostream>

long nwp(long p, long q); // najwiekszy wspólny podzielnik

class Fraction;
typedef const Fraction cFraction; // Skrót notacyjny

class Fraction
{
  long l;
  unsigned long m; // Mianownik nieujemny
  void initFraction(long a, long b);
public:
  Fraction(long ll = 0, long mm = 1L) // Konstruktor (3 wersje )
  {
    initFraction(ll, mm);
  }
  long licznik() const { return l; }
  long mianownik() const { return m; }
  Fraction add(cFraction& a);
  Fraction sub(cFraction& a);
  void show();
};
```

Klasa Point (plik: Fraction.cpp)

```
#include "Fraction.h"
#include <iostream>
using std::ostream;
using std::istream;
using std::cout;
using std::endl;

long nwp(long p, long q) // najwiekszy wspólny podzielnik
{

  if (p < 0) { p = -p; }
  if (q < 0) { q = -q; }

  if (p == 0 && q == 0) return 1;
  if (p == 0) return q;
  if (q == 0) return p;

  while (p != q)
  {
    if (p > q) {
      p = p - q;
    }
    else {
      q = q - p;
    }
  }

  return p;
}
…
```

Klasa Point (plik: Fraction.cpp)

```
…
void Fraction::initFraction(long ll, long mm)
{
  long d = nwp(ll, mm);
  if (mm < 0) { ll = -ll; mm = -mm; } // Mianownik zawsze nieujemny
  l = ll / d; m = mm / d;
}

Fraction Fraction::add(cFraction& a)
{
  long tl, tm;
  if (m == 0 && a.m == 0) { tl = (l + a.l) / 2; tm = 0; }
  else { tl = l * a.m + m * a.l; tm = m * a.m; }
  return Fraction(tl, tm);
}
Fraction Fraction::sub(cFraction& a)
{
  long tl, tm;
  if (m == 0 && a.m == 0) { tl = (l - a.l) / 2; tm = 0; }
  else { tl = l * a.m - m * a.l; tm = m * a.m; }
  return Fraction(tl, tm);
}

void Fraction::show()
{
  cout << l;
  if (m != 1) cout << "/" << m;
}
```

Klasa Point (plik: main.cpp)

```
#include <iostream>
#include "Fraction.h"
#include <iostream>

using std::cout;
using std::endl;

//#define P(x) cout <<#x" = " <<(x)<<endl // Makro do wydruków

int main()
{
  Fraction f0, f1(1, 2), f2(2, 3), f3(3, 4);
  Fraction f4 = f1.add(f2);
  Fraction f5 = f3.sub(f1);

  f1.show(); cout << endl;
  f2.show(); cout << endl;
  f3.show(); cout << endl;
  f4.show(); cout << endl;
  f5.show(); cout << endl;


  return 0;
}
```

Klasa Fraction – podejście drugie

Przeciążanie operatorów **+** i **-** przy wykorzystaniu metod.

Nie można przeciążać operatorów

Operator	Description
sizeof	The sizeof operator
.	The membership operator
.*	The pointer-to-member operator
::	The scope-resolution operator
?:	The conditional operator
typeid	An RTTI operator
const_cast	A type cast operator
dynamic_cast	A type cast operator
reinterpret_cast	A type cast operator
static_cast	A type cast operator

Operatory, których przeciążanie jest możliwe jedynie metodą

Operator	Description
=	Assignment operator
()	Function call operator
[]	Subscripting operator
->	Class member access by pointer operator

Operatory zdatne do przeciążania zarówno metodą jak i funkcją zaprzyjaźnioną

+	-	*	/	%	^
&	|	~	!	=	<
>	+=	-=	*=	/=	%=
^=	&=	|=	<<	>>	>>=
<<=	==	!=	<=	>=	&&
||	++	--	,	->*	->
()	[]	new	delete	new []	delete []

Klasa Point (plik: Fraction.h)

```
#pragma once
#include <iostream>

long nwp(long p, long q); // najwiekszy wspólny podzielnik

class Fraction;
typedef const Fraction cFraction; // Skrót notacyjny

class Fraction
{
  long l;
  unsigned long m; // Mianownik nieujemny
  void initFraction(long a, long b);
public:
  Fraction(long ll = 0, long mm = 1L) // Konstruktor (3 wersje )
  {
    initFraction(ll, mm);
  }
  long licznik() const { return l; }
  long mianownik() const { return m; }
  Fraction operator+(cFraction& a);
  Fraction operator-(cFraction& a);
  void show();
};
```

Klasa Point (plik: Fraction.cpp)

```
…
void Fraction::initFraction(long ll, long mm)
{
  long d = nwp(ll, mm);
  if (mm < 0) { ll = -ll; mm = -mm; } // Mianownik zawsze nieujemny
  l = ll / d; m = mm / d;
}

// Operatory uwzględniają "wartości" 1/0, -1/0, 0/0
Fraction Fraction::operator+(cFraction& a)
{
  long tl, tm;
  if (m == 0 && a.m == 0) { tl = (l + a.l) / 2; tm = 0; }
  else { tl = l * a.m + m * a.l; tm = m * a.m; }
  return Fraction(tl, tm);
}
Fraction Fraction::operator-(cFraction& a)
{
  long tl, tm;
  if (m == 0 && a.m == 0) { tl = (l - a.l) / 2; tm = 0; }
  else { tl = l * a.m - m * a.l; tm = m * a.m; }
  return Fraction(tl, tm);
}

void Fraction::show()
{
  cout << l;
  if (m != 1) cout << "/" << m;
}
```

Klasa Point (plik: main.cpp)

```
#include <iostream>
#include "Fraction.h"
#include <iostream>

using std::cout;
using std::endl;

//#define P(x) cout <<#x" = " <<(x)<<endl // Makro do wydruków

int main()
{
  Fraction f0, f1(1, 2), f2(2, 3), f3(3, 4);
  Fraction f4 = f1 + f2;
  Fraction f5 = f3 - f1;

  f1.show(); cout << endl;
  f2.show(); cout << endl;
  f3.show(); cout << endl;
  f4.show(); cout << endl;
  f5.show(); cout << endl;


  return 0;
}
```

Klasa Fraction – podejście trzecie

Przeciążanie operatorów przy wykorzystaniu **funkcji zaprzyjaźnionych**.

Klasa Point (plik: Fraction.h)

```
#pragma once
#include <iostream>

long nwp(long p, long q); // najwiekszy wspólny podzielnik

class Fraction;
typedef const Fraction cFraction; // Skrót notacyjny

class Fraction
{
  long l;
  unsigned long m; // Mianownik nieujemny
  void initFraction(long a, long b);
public:
  Fraction(long ll = 0, long mm = 1L) // Konstruktor (3 wersje )
  {
    initFraction(ll, mm);
  }
  long licznik() const { return l; }
  long mianownik() const { return m; }
  friend Fraction operator+(cFraction& a, cFraction& b);
  friend Fraction operator-(cFraction& a, cFraction& b);
  friend Fraction operator*(cFraction& a, cFraction& b);
  friend Fraction operator/(cFraction& a, cFraction& b);
  // ...
  friend bool operator<(cFraction& a, cFraction& b);
  friend bool operator>(cFraction& a, cFraction& b);
  friend bool operator<=(cFraction& a, cFraction& b);
  friend bool operator>=(cFraction& a, cFraction& b);
  friend bool operator==(cFraction& a, cFraction& b);
  friend bool operator!=(cFraction& a, cFraction& b);
  void show();
};
```

Klasa Point (plik: Fraction.cpp)

```
…
// Operatory uwzględniają "wartości" 1/0, -1/0, 0/0
Fraction operator+(cFraction& a, cFraction& b)
{
  long tl, tm;
  if (a.m == 0 && b.m == 0) { tl = (a.l + b.l) / 2; tm = 0; }
  else { tl = a.l * b.m + a.m * b.l; tm = a.m * b.m; }
  return Fraction(tl, tm);
}
Fraction operator-(cFraction& a, cFraction& b)
{
  long tl, tm;
  if (a.m == 0 && b.m == 0) { tl = (a.l - b.l) / 2; tm = 0; }
  else { tl = a.l * b.m - a.m * b.l; tm = a.m * b.m; }
  return Fraction(tl, tm);
}
Fraction operator*(cFraction& a, cFraction& b)
{
  long d1, d2;
  d1 = nwp(a.l, b.m); // Podzielniki wzajemne a i b
  d2 = nwp(a.m, b.l);
  return Fraction((a.l / d1) * (b.l / d2), (a.m / d2) * (b.m / d1));
}
…
```

Klasa Point (plik: Fraction.cpp)

```
…
Fraction operator/(cFraction& a, cFraction& b)
{
  long tl, tm;
  if (a.m == 0 && b.m == 0) {
    tl = (a.l - b.l) / 2;
    tm = 0;
    return Fraction(tl, tm);
  }


  long d1, d2;
  d1 = nwp(a.l, b.l); // Podzielniki wzajemne a i b
  d2 = nwp(a.m, b.m);
  return Fraction((a.l / d1) * (b.m / d2), (a.m / d2) * (b.l / d1));
}
// Wybrane operatory relacji
bool operator< (cFraction& a, cFraction& b) { return (a - b).l < 0; }
bool operator>=(cFraction& a, cFraction& b) { return !(a < b); }
bool operator==(cFraction& a, cFraction& b) { return (a - b).l == 0; }
bool operator!=(cFraction& a, cFraction& b) { return !(a == b); }
void Fraction::show()
{
  cout << l;
  if (m != 1) cout << "/" << m;
}
```

Klasa Point (plik: main.cpp)

```cpp
#include <iostream>
#include "Fraction.h"
#include <iostream>

using std::cout;
using std::endl;

#define P(x) cout <<#x" = " <<(x)<<endl // Makro do wydruków

int main()
{
  Fraction f0, f1(1, 2), f2(2, 3), f3(3, 4), ft;

  f0.show(); cout << endl; // f0 = 0
  f1.show(); cout << endl; // f1 = 1/2
  f2.show(); cout << endl; // f2 = 2/3
  f3.show(); cout << endl; // f3 = 3/4

  ft = ((f1 + f2) / (f3 - f1)); // (f1+f2)/(f3 -f1) = 14/3
  ft.show(); cout << endl;
  ft = (1 / f3 + 1 / f2); // 1/f3 + 1/f2 = 17/6
  ft.show(); cout << endl;
  ft = (12 / f0 + -12 / f0); // 12/ f0 + -12/f0 = 0/0
  ft.show(); cout << endl;
  ft = (12 / f0); // 12/ f0 = 1/0
  ft.show(); cout << endl;
  ft = (-12 / f0); // -12/f0 = -1/0
  ft.show(); cout << endl;
  ft = (3.0 * f3 / f2); // 3.0*f3/f2 = 27/8
  ft.show(); cout << endl;

  return 0;
}
```

Klasa Fraction – podejście czwarte

Przeciążanie operatorów **<<** i **>>**.

Klasa Point (plik: Fraction.h)

```
#pragma once
#include <iostream>

long nwp(long p, long q); // najwiekszy wspólny podzielnik

class Fraction;
typedef const Fraction cFraction; // Skrót notacyjny

class Fraction
{
  long l;
  unsigned long m; // Mianownik nieujemny
  void initFraction(long a, long b);
public:
  Fraction(long ll = 0, long mm = 1L) // Konstruktor (3 wersje )
  {
    initFraction(ll, mm);
  }
  long licznik() const { return l; }
  long mianownik() const { return m; }
  friend Fraction operator+(cFraction& a, cFraction& b);
  friend Fraction operator-(cFraction& a, cFraction& b);
  friend Fraction operator*(cFraction& a, cFraction& b);
  friend Fraction operator/(cFraction& a, cFraction& b);
  // ...
  friend bool operator<(cFraction& a, cFraction& b);
  friend bool operator>(cFraction& a, cFraction& b);
  friend bool operator<=(cFraction& a, cFraction& b);
  friend bool operator>=(cFraction& a, cFraction& b);
  friend bool operator==(cFraction& a, cFraction& b);
  friend bool operator!=(cFraction& a, cFraction& b);
  friend std::ostream& operator<< (std::ostream& os, cFraction& f);
  friend std::istream& operator>> (std::istream& is, Fraction& f);
};
```

Klasa Point (plik: Fraction.cpp)

```
…
// Operatory wejścia - wyjścia

ostream& operator<<(ostream& os, cFraction& f)
{
  os << f.l;
  if (f.m != 1) cout << "/" << f.m;
  return os;
}

istream& operator>>(istream& is, Fraction& f)
{
  is >> f.l; // Licznik pobierany zawsze
  f.m = 1; // Tymczasowo
  char c = is.get();
  if (c == '/') // Czytaj mianownik
    is >> f.m;
  else
    is.unget(); // Wycofaj znak nie dla nas
  f.initFraction(f.l, f.m); // Wymuszenie niezmiennika
  return is;
}
```

Klasa Point (plik: main.cpp)

```cpp
#include <iostream>
#include "Fraction.h"
#include <iostream>

using std::cin;
using std::cout;
using std::endl;

#define P(x) cout <<#x" = " <<(x)<<endl // Makro do wydruków

int main()
{
  Fraction f0, f1(1, 2), f2(2, 3), f3(3, 4);

  P(f0); // f0 = 0
  P(f1); // f1 = 1/2
  P(f2); // f2 = 2/3
  P(f3); // f3 = 3/4
  P((f1 + f2) / (f3 - f1)); // (f1+f2)/(f3 -f1) = 14/3
  P(1 / f3 + 1 / f2); // 1/f3 + 1/f2 = 17/6
  P(12 / f0 + -12 / f0); // 12/ f0 + -12/f0 = 0/0
  P(12 / f0); // 12/ f0 = 1/0
  P(-12 / f0); // -12/f0 = -1/0
  P(3.0 * f3 / f2); // 3.0*f3/f2 = 27/8
  //cout << "koniec" << endl;

  Fraction ff;
  cout << "wprowadz ulamek: ";
  cin >> ff;
  cout << "wprowadzony przez Ciebie ulamek to: " << ff << endl;

  return 0;
}
```

Przeciążanie operatorów

Podsumowanie :

- Wykorzystanie metod do przeprowadzania operacji arytmetycznych
- Przeciążanie operatorów – implementacja wykorzystująca metody
- Przeciążanie operatorów – implementacja wykorzystująca funkcje
- Przeciążanie operatorów **<<** i **>>**

Dynamiczny przydział pamięci

Dynamiczny przydział pamięci

Plan zajęć :

- Operatory **new** i **delete**
- Stosowanie dynamicznego przydziału pamięci dla składowych klas
- Niejawne i jawne konstruktory kopiujące
- Niejawne i jawne przeciążone operatory przypisania
- Obowiązki wynikające ze stosowania **new** w konstruktorach

Operatory new i delete

```
int* pn = new int;

int higgens;
int* pt = &higgens;

typeName* pointer_name = new typeName;

int* ps = new int; // allocate memory with new
. . . // use the memory
delete ps; // free memory with delete when done

int* ps = new int; // ok
delete ps; // ok
delete ps; // not ok now
int jugs = 5; // ok
int* pi = &jugs; // ok
delete pi; // not allowed, memory not allocated by new
```

Operatory new i delete

```
int* psome = new int[10]; // get a block of 10 ints
...
delete[] psome; // free a dynamic array


int* pt = new int;
short* ps = new short[500];
delete[] pt; // effect is undefined, don't do it
delete ps; // effect is undefined, don't do it
```

Operatory new i delete

```
int* psome = new int[10]; // get a block of 10 ints
...
delete[] psome; // free a dynamic array


int* pt = new int;
short* ps = new short[500];
delete[] pt; // effect is undefined, don't do it
delete ps; // effect is undefined, don't do it
```

Operatory new i delete

In short, you should observe these rules when you use new and delete:

- Don’t use delete to free memory that new didn’t allocate.
- Don’t use delete to free the same block of memory twice in succession.
- Use delete [] if you used new [] to allocate an array.
- Use delete (no brackets) if you used new to allocate a single entity.
- It’s safe to apply delete to the null pointer (nothing happens).

Special Member Functions

In particular, C++ automatically provides the following member functions:

- A default constructor if you define no constructors
- A default destructor if you don’t define one
- A copy constructor if you don’t define one
- An assignment operator if you don’t define one
- An address operator if you don’t define one

Copy Constructors

A copy constructor is used to copy an object to a newly created object.That is, it's used during initialization, including passing function arguments by value and not during ordinary assignment. A copy constructor for a class normally has this prototype:

```
Class_name(const Class_name&);
```

Note that it takes a constant reference to a class object as its argument.

You must know two things about a copy constructor: when it's used and what it does.

When a Copy Constructor Is Used

A copy constructor is invoked whenever a new object is created and initialized to an existing object of the same kind. This happens in several situations.The most obvious situation is when you explicitly initialize a new object to an existing object. For example, given that motto is a StringBad object, the following four defining declarations invoke a copy constructor:

```
StringBad ditto(motto);
// calls StringBad(const StringBad &)
StringBad metoo = motto;
// calls StringBad(const StringBad &)
StringBad also = StringBad(motto);
// calls StringBad(const StringBad &)
StringBad* pStringBad = new StringBad(motto);
// calls StringBad(const StringBad &)
```

When a Copy Constructor Is Used

Less obviously, a compiler uses a copy constructor whenever a program generates copies of an object. In particular, it's used when

- a function passes an object by value
- or when a function returns an object

Remember, passing by value means creating a copy of the original variable. A compiler also uses a copy constructor whenever it generates temporary objects. For example, a compiler might generate a temporary Vector object to hold an intermediate result when adding three Vector objects.

What a Default Copy Constructor Does

The default copy constructor performs a member-by-member copy of the nonstatic members (memberwise copying, also sometimes called shallow copying). Each member is copied by value.

Fixing the Problem by Defining an Explicit Copy Constructor

The cure for the problems in the class design is to make a *deep copy*.

Assignment Operators

This operator has the following prototype:

```
Class_name& Class_name::operator=(const Class_name&);
```

That is, it takes and returns a reference to an object of the class.

When an Assignment Operator Is Used and What It Does

An overloaded assignment operator is used when you assign one object to another existing

object:

```
StringBad headline1("Celery Stalks at Midnight");
...
StringBad knot;
knot = headline1; // assignment operator invoked
```

An assignment operator is not necessarily used when initializing an object:

```
StringBad metoo = knot; // use copy constructor, possibly
assignment, too
```

When an Assignment Operator Is Used and What It Does

Like a copy constructor, an implicit implementation of an assignment operator performs a member-to-member copy. If a member is itself an object of some class, the program uses the assignment operator defined for that class to do the copying for that particular member. Static data members are unaffected.

Things to Remember When Using new in Constructors

By now you've noticed that you must take special care when using new to initialize pointer members of an object. In particular, you should do the following:

- If you use new to initialize a pointer member in a constructor, you should use delete in the destructor.
- The uses of new and delete should be compatible. You should pair new with delete and new [] with delete [].
- If there are multiple constructors, all should use new the same way—either all with brackets or all without brackets. There's only one destructor, so all constructors have to be compatible with that destructor.

Dynamiczny przydział pamięci

Podsumowanie :

- Operatory **new** i **delete**
- Stosowanie dynamicznego przydziału pamięci dla składowych klas
- Niejawne i jawne konstruktory kopiujące
- Niejawne i jawne przeciążone operatory przypisania
- Obowiązki wynikające ze stosowania **new** w konstruktorach

Dziedziczenie

Dziedziczenie

Plan zajęć :

- Dziedziczenie proste
- Dziedziczenie polimorficzne
- Abstrakcyjne klasy bazowe
- Dziedziczenie i dynamiczny przydział pamięci

Dziedziczenie

Korzyści płynące z dziedziczenia:

- Dziedziczenie pozwala dodawać nowe możliwości do istniejącej klasy.
- Dziedziczenie pozwala dodawać nowe dane do klasy.
- Dziedziczenie pozwala zmieniać działanie metod klasy.

Dziedziczenie proste

Dziedziczenie:

- Obiekt typu pochodnego zawiera w sobie dane składowe typu bazowego. Klasa pochodna dziedziczy implementację klasy bazowej.
- Obiekt typu pochodnego może używać metod typu bazowego. Klasa pochodna dziedziczy interfejs klasy bazowej.

Co trzeba dodać do odziedziczonych cech?

- Klasa pochodna potrzebuje własnego konstruktora.
- W klasie pochodnej można dodać nowe dane składowe oraz metody.

W przypadku dziedziczenia prostego obiekty klasy pochodnej używają metod klasy bazowej w ich oryginalnej formie.

Dziedziczenie proste

Konstruktor:

- Kiedy program tworzy obiekt klasy pochodnej, najpierw wywołuje konstruktor klasy bazowej, a następnie konstruktor klasy pochodnej.
- Konstruktor klasy bazowej jest odpowiedzialny za inicjalizację odziedziczonych danych składowych.
- Konstruktor klasy pochodnej jest odpowiedzialny za inicjalizację wszystkich nowych danych składowych.
- Konstruktor klasy pochodnej zawsze wywołuje konstruktor klasy bazowej.
- Można użyć listy inicjalizacyjnej, aby określić, który konstruktor klasy bazowej ma zostać wywołany.

Destruktor:

- Kiedy obiekt klasy pochodnej jest usuwany, program najpierw wywołuje destruktor klasy pochodnej, a następnie destruktor klasy bazowej.

Dziedziczenie polimorficzne

Kiedy dziedziczenie proste nie wystarcza?

- W sytuacji, w której metody powinny zachowywać się inaczej dla obiektów klasy pochodnej niż dla obiektów klasy bazowej.

Istnieją dwa główne mechanizmy związane z implementacją polimorficznego dziedziczenia publicznego:

- Ponowne definiowanie metody klasy bazowej w klasie pochodnej.
- Używanie metod wirtualnych.

W przypadku dziedziczenia polimorficznego obiekty klasy pochodnej mogą używać metod klasy bazowej w zmodyfikowanej formie.

Polimorficzne – „mające wiele postaci”

Abstrakcyjne klasy bazowe

- Abstrakcyjna klasa bazowa definiuje interfejs bez wgłębiania się w kwestie związane z implementacją.
- Abstrakcyjna klasa bazowa musi zawierać przynajmniej jedną funkcję czysto wirtualną.
- Funkcję czysto wirtualną deklaruje się umieszczając = 0przed średnikiem który kończy wyrażenie:

```
virtual double Area() const = 0;
```

Abstrakcyjne klasy bazowe służą do zdefiniowania wspólnego interfejsu dla klas pochodnych.

Dziedziczenie i dynamiczny przydział pamięci

Klasa pochodna bez dynamicznego przydziału pamięci:

- Nie trzeba podejmować żadnych specjalnych kroków.

Klasa pochodna z dynamicznym przydziałem pamięci:

- W klasie pochodnej trzeba zdefiniować destruktor, konstruktor kopiujący oraz operator przypisania.

Dziedziczenie

Podsumowanie :

- Dziedziczenie proste
- Dziedziczenie polimorficzne
- Abstrakcyjne klasy bazowe
- Dziedziczenie i dynamiczny przydział pamięci

Agregacja

Agregacja

Plan zajęć :

- Zawieranie
- Dziedziczenie prywatne

Klasa valarray

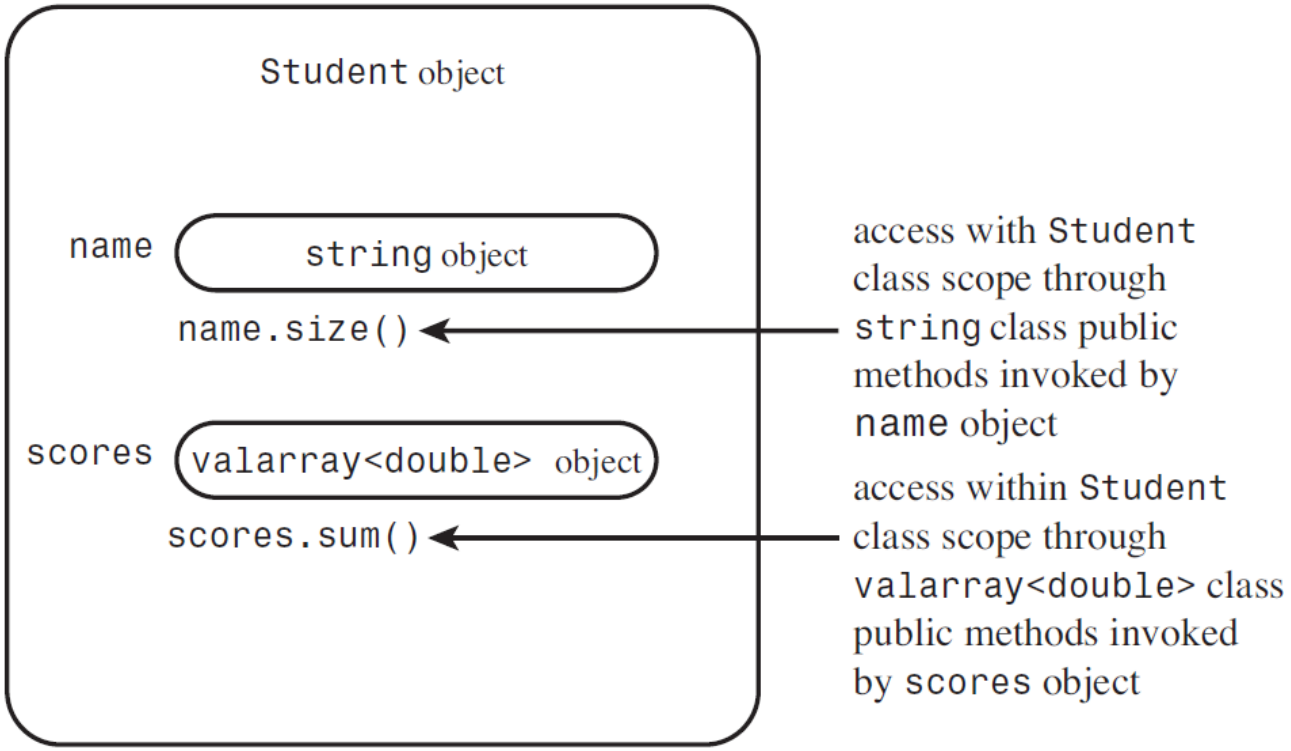
```
using std::valarray;
valarray<int> q_values;
valarray<double> weights;

double gpa[5] = { 3.1, 3.5, 3.8, 2.9, 3.3 };
valarray<double> v1; // Tablica liczb zmiennoprzecinkowych,
0 elementów
valarray<int> v2(8); // Tablica liczb całkowitych, 8
elementów
valarray<int> v3(10, 8); //Tablica liczb całkowitych, 8
elementów,
// każdy o wartości 10
valarray<double> v4(gpa, 4); //Tablica 4 elementów
// zainicjalizowana za pomocą czterech pierwszych
// elementów tablicy gpa
```

Klasa valarray

Lista wybranych metod klasy **valarray**:

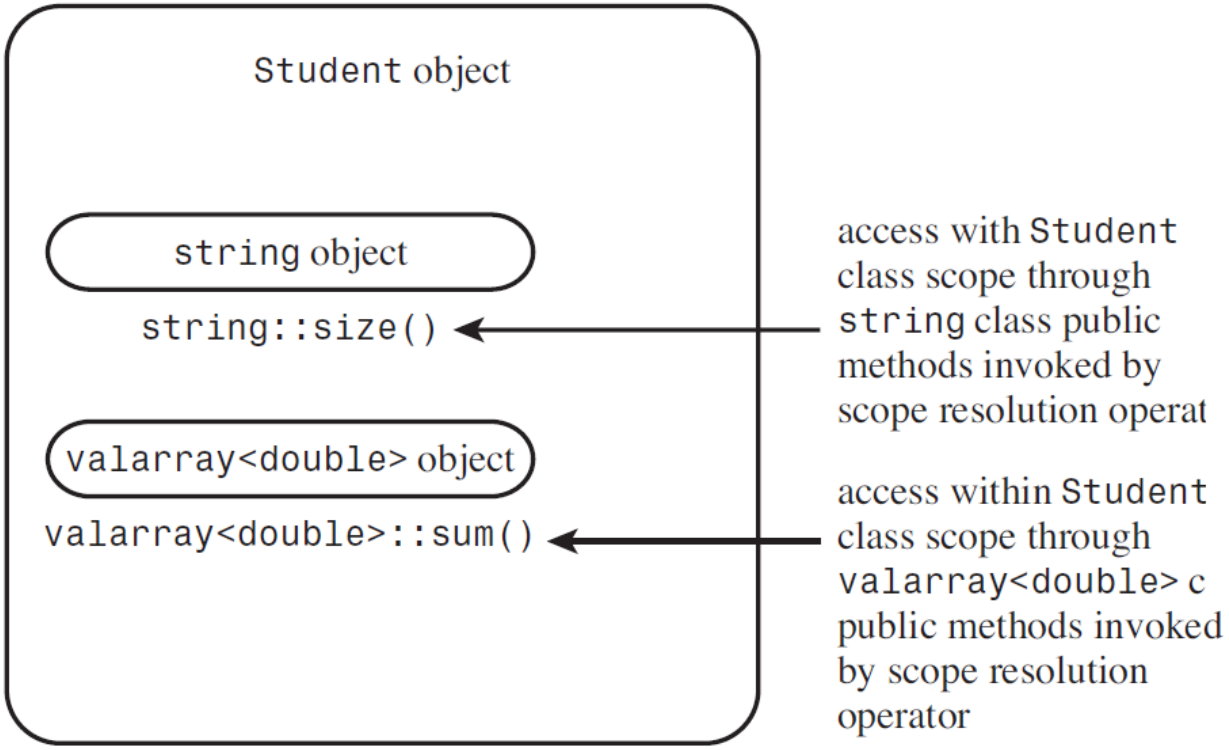
- Funkcja **operator** [ ] () umożliwia dostęp do konkretnego elementu.
- Funkcja **size** () zwraca liczbę elementów.
- Funkcja **sum** () zwraca sumę elementów.
- Funkcja **max** () zwraca największy element.
- Funkcja **min** () zwraca najmniejszy element.

Zawieranie




```
class Student
{
private
    string name;
    valarray<double> scores;
    …
};
```

Dziedziczenie prywatne




```
class Student:private string,
private valarray<double>
{
   …
};
```

Zawieranie a dziedziczenie prywatne

- Łatwiejsza analiza kodu (widoczne obiekty o znaczących nazwach reprezentujące zawierane klasy)
- Dziedziczenie powoduje że relacja wygląda bardziej abstrakcyjnie
- Dziedziczenie może powodować problemy, szczególnie gdy klasa dziedziczy po więcej niż jednej klasie bazowej. W niektórych sytuacjach trzeba zmagać się z różnymi klasami które mają wspólnego przodka lub metody o takiej samej nazwie.
- Zawieranie umożliwia umieszczenie w klasie więcej niż jednego pod-obiektu tej samej klasy. Dziedziczenie ogranicza nas do pojedynczego obiektu (trudno określić różne obiekty, jeśli nie mają one nazw).

Zawieranie a dziedziczenie prywatne

Do modelowania relacji ***ma-coś*** zwykle powinno się używać **zawierania**.

Dziedziczenia prywatnego używamy wtedy, kiedy nowa klasa potrzebuje **dostępu do składowych chronionych** w oryginalnej klasie lub zachodzi potrzeba **redefinicji funkcji wirtualnych**.

Rodzaje dziedziczenia

Property	Public Inheritance	Protected Inheritance	Private Inheritance
Public members become	Public members of the derived class	Protected members of the derived class	Private members of the derived class
Protected members become	Protected members of the derived class	Protected members of the derived class	Private members of the derived class
Private members become	Accessible only through the base-class interface	Accessible only through the base-class interface	Accessible only through the base-class interface
Implicit upcasting	Yes	Yes (but only the derived class) within	No

Agregacja

Podsumowanie :

- Zawieranie
- Dziedziczenie prywatne

Obsługa błędów – wyjątki

Obsługa błędów – wyjątki

Plan zajęć :

- Zwracanie kodu błędu
- Mechanizm wyjątków
- Wyjątki w postaci obiektów
- Rozwijanie stosu (różnice między throw a return)
- Przekazanie wyjątku przez wartość i przez referencję
- Klasa exception

stosowanie funkcji abort()

```
#include <iostream>
#include <cstdlib>
double hmean(double a, double b);

int main()
{
  double x, y, z;

  std::cout << "Podaj dwie liczby: ";
  while (std::cin >> x >> y)
  {
    z = hmean(x, y);
    std::cout << "Średnia harmoniczna liczb " << x << " i " << y
      << " wynosi " << z << std::endl;
    std::cout << "Podaj kolejną parę liczb <w, aby wyjść>: ";
  }
  std::cout << "Koniec\n";
  return 0;
}

double hmean(double a, double b)
{
  if (a == -b)
  {
    std::cout << "Niepoprawne argumenty funkcji hmean()\n";
    std::abort();
  }
  return 2.0 * a * b / (a + b);
}
```

zwracanie kodu błędu

```
#include <iostream>
#include <cfloat> // (lub float.h) dla DBL_MAX

bool hmean(double a, double b, double* ans);

int main()
{
  double x, y, z;

  std::cout << "Podaj dwie liczby: ";
  while (std::cin >> x >> y)
  {
    if (hmean(x, y, &z))
      std::cout << "Średnia harmoniczna liczb " << x << " i " << y
      << " wynosi " << z << std::endl;
    else
      std::cout << "Suma liczb nie może wynosić 0 -"
      << " spróbuj jeszcze raz.\n";
    std::cout << "Podaj kolejną parę liczb <w, aby wyjść>: ";
  }
  std::cout << "Koniec\n";
  return 0;
}

bool hmean(double a, double b, double* ans)
{
  if (a == -b)
  {
    *ans = DBL_MAX;
    return false;
  }
  else
  {
    *ans = 2.0 * a * b / (a + b);
    return true;
  }
}
```

używanie wyjątków

```
#include <iostream>
double hmean(double a, double b);

int main()
{
  double x, y, z;

  std::cout << "Podaj dwie liczby: ";
  while (std::cin >> x >> y)
  {
    try {        // początek bloku try
      z = hmean(x, y);
    }       // koniec bloku try
    catch (const char* s)   // początek bloku catch
    {
      std::cout << s << std::endl;
      std::cout << "Podaj kolejną parę liczb: ";
      continue;
    }    // koniec bloku catch
    std::cout << "Średnia harmoniczna liczb " << x << " i " << y
      << " wynosi " << z << std::endl;
    std::cout << "Podaj kolejną parę liczb <w, aby wyjść>: ";
  }
  std::cout << "Koniec\n";
  return 0;
}

double hmean(double a, double b)
{
  if (a == -b)
    throw "Niepoprawne argumenty funkcji hmean(): a = -b nie jest
dozwolone";
  return 2.0 * a * b / (a + b);
}
```

używanie klasy wyjątków

```
#include <iostream>
#include <cmath>
#include "exc_mean.h"
// prototypy funkcji
double hmean(double a, double b);
double gmean(double a, double b);
int main()
{
  using std::cout;
  using std::cin;
  using std::endl;

  double x, y, z;

  cout << "Podaj dwie liczby: ";
  while (cin >> x >> y)
  {
    try {      // początek bloku try
      z = hmean(x, y);
      cout << "Średnia harmoniczna liczb " << x << " i " << y
        << " wynosi " << z << endl;
      cout << "Średnia geometryczna liczb " << x << " i " << y
        << " wynosi " << gmean(x, y) << endl;
      cout << "Podaj kolejną parę liczb <w, aby wyjść>: ";
    }   // koniec bloku try
    catch (bad_hmean & bg)   // początek bloku catch
    {
      bg.mesg();
      cout << "Spróbuj ponownie.\n";
      continue;
    }
    catch (bad_gmean & hg)
    {
      cout << hg.mesg();
      cout << "Użyte wartości: " << hg.v1 << ", "
        << hg.v2 << endl;
      cout << "Niestety, to koniec zabawy.\n";
      break;
    } // koniec bloku catch
  }
  cout << "Koniec\n";
  return 0;
}
```

```
double hmean(double a, double b)
{
  if (a == -b)
    throw bad_hmean(a, b);
  return 2.0 * a * b / (a + b);
}

double gmean(double a, double b)
{
  if (a < 0 || b < 0)
    throw bad_gmean(a, b);
  return std::sqrt(a * b);
}
```

używanie klasy wyjątków

```
#include <iostream>

class bad_hmean
{
private:
  double v1;
  double v2;
public:
  bad_hmean(double a = 0, double b = 0) : v1(a), v2(b) {}
  void mesg();
};

inline void bad_hmean::mesg()
{
  std::cout << "hmean(" << v1 << ", " << v2 << "): "
    << "niepoprawne argumenty: a = -b\n";
}

class bad_gmean
{
public:
  double v1;
  double v2;
  bad_gmean(double a = 0, double b = 0) : v1(a), v2(b) {}
  const char* mesg();
};

inline const char* bad_gmean::mesg()
{
  return "Argumenty funkcji gmean() powinny być >= 0\n";
}
```

rozwijanie stosu

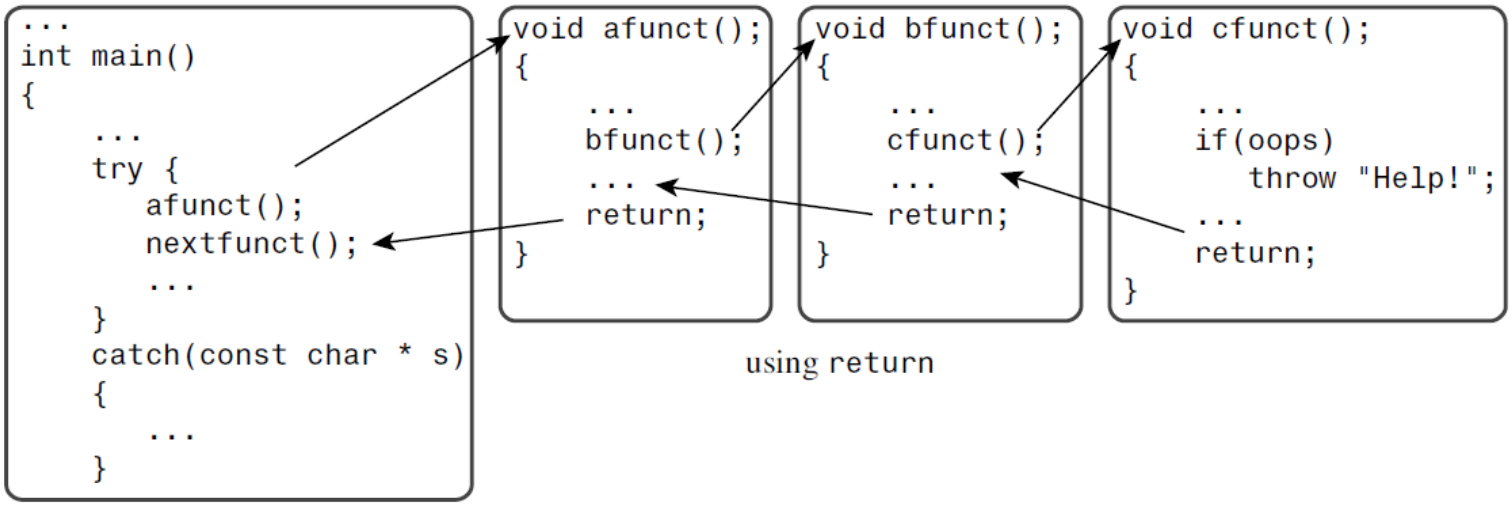
```
#include <iostream>
#include <cmath> // lub math.h; użytkownicy systemu UNIX mogą
potrzebować opcji -lm
#include <string>
#include "exc_mean.h"

class demo
{
private:
  std::string word;
public:
  demo(const char* str)
  {
    word = str;
    std::cout << "Obiekt demo " << word << " utworzony\n";
  }
  ~demo()
  {
    std::cout << "Obiekt demo " << word << " usunięty\n";
  }
  void show() const
  {
    std::cout << "Obiekt demo " << word << " żyje\n";
  }
};
```

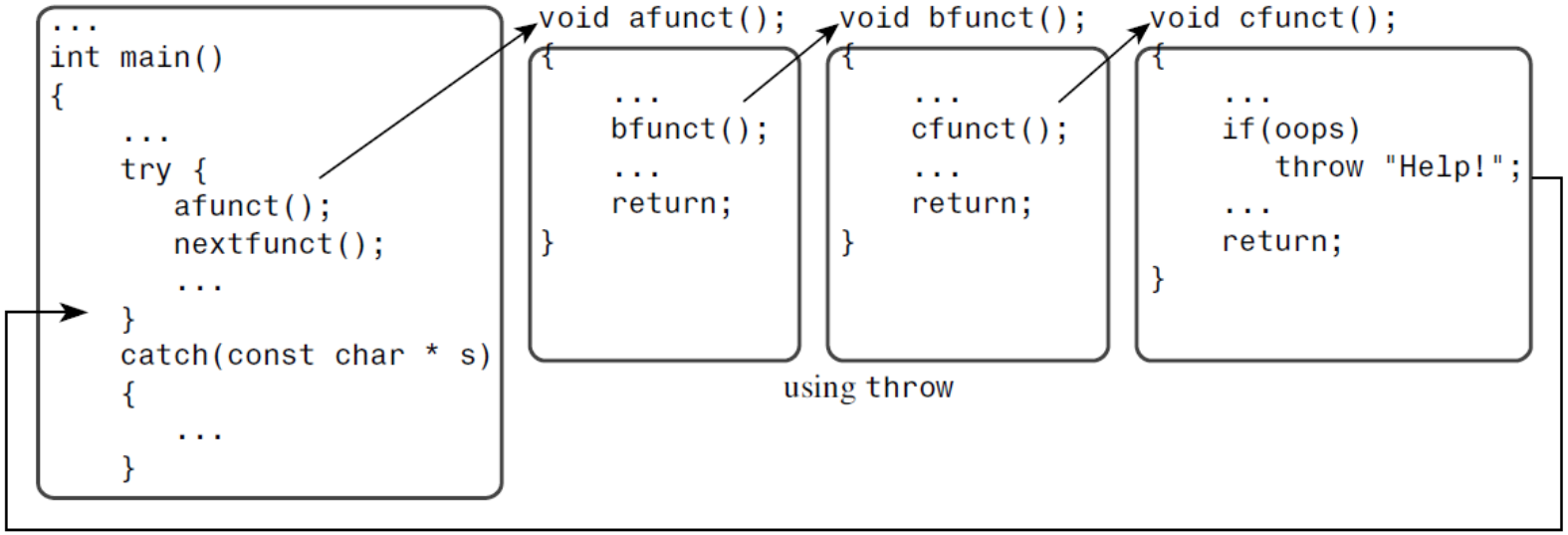
rozwijanie stosu

```
double hmean(double a, double b);
double gmean(double a, double b);
double means(double a, double b);

int main()
{
  setlocale(LC_ALL, "pl_PL");
  using std::cout;
  using std::cin;
  using std::endl;

  double x, y, z;
  {
    demo d1("z bloku zagnieżdżonego w funkcji main()");
    cout << "Podaj dwie liczby: ";
    while (cin >> x >> y)
    {
      try {   // początek bloku try
        z = means(x, y);
        cout << "Średnia średnich liczb " << x << " i " << y
          << " wynosi " << z << endl;
        cout << "Podaj kolejną parę liczb: ";
      }   // koniec bloku try
      catch (bad_hmean & bg) // początek bloku catch
      {
        bg.mesg();
        cout << "Spróbuj ponownie.\n";
        continue;
      }
      catch (bad_gmean & hg)
      {
        cout << hg.mesg();
        cout << "Użyte wartości: " << hg.v1 << ", "
          << hg.v2 << endl;
        cout << "Niestety, to koniec zabawy.\n";
        break;
      } // koniec bloku catch
    }
    d1.show();
  }
  cout << "Koniec\n";
  cin.get();
  cin.get();
  return 0;
}
```

```
double hmean(double a, double b)
{
  if (a == -b)
    throw bad_hmean(a, b);
  return 2.0 * a * b / (a + b);
}

double gmean(double a, double b)
{
  if (a < 0 || b < 0)
    throw bad_gmean(a, b);
  return std::sqrt(a * b);
}

double means(double a, double b)
{
  double am, hm, gm;
  demo d2("z funkcji means()");
  am = (a + b) / 2.0; // średnia arytmetyczna
  try
  {
    hm = hmean(a, b);
    gm = gmean(a, b);
  }
  catch (bad_hmean & bg) // początek bloku catch
  {
    bg.mesg();
    std::cout << "Przechwycony w means()\n";
    throw; // ponownie zgłasza wyjątek
  }
  d2.show();
  return (am + hm + gm) / 3.0;
}
```

rozwijanie stosu





rozwijanie stosu

- Although the throw-catch mechanism is similar to function arguments and the function return mechanism, there are a few differences. One, which you've already encountered, is that a return statement in a function fun() transfers execution to the function that called fun(), but a throw transfers execution all the way up to the first function having a try-catch combination that catches the exception.
- Another difference is that the compiler always creates a temporary copy when throwing an exception, even if the exception specifier and catch blocks specify a reference.

```
class problem { ... };
...
void super() throw (problem)
{
  ...
    if (oh_no)
    {
      problem oops; // construct object
      throw oops; // throw it
      ...
    }
  ...
    try {
    super();
  }
  catch (problem & p)
  {
    // statements
  }
```

przekazanie wyjątku przez wartość i przez referencję

```
class bad_1 { ... };
class bad_2 : public bad_1 { ... };
class bad_3 : public bad 2 {...};
...
void duper()
{
  ...
    if (oh_no)
      throw bad_1();
  if (rats)
    throw bad_2();
  if (drat)
    throw bad_3();
}
...
try {
  duper();
}
catch (bad_3 & be)
{ // statements }
catch (bad_2 & be)
{ // statements }
catch (bad_1 & be)
{ // statements }
```

Tip

If you have an inheritance hierarchy of exception classes, you should arrange the order of the catch blocks so that the exception of the most-derived class (that is, the class furthest down the class hierarchy sequence) is caught first and the base-class exception is caught last.

klasa exception

```
#include <exception>
class bad_hmean : public std::exception
{
public:
  const char* what() { return "bad arguments to hmean()"; }
  ...
};
class bad_gmean : public std::exception
{
public:
  const char* what() { return "bad arguments to gmean()"; }
  ...
};


try {
  ...
}
catch (std::exception & e)
{
  cout << e.what() << endl;
  ...
}
```

klasa stdexcept

The stdexcept header file defines several more exception classes. First, the file defines the logic_error and runtime_error classes, both of which derive publicly from exception:

```
class logic_error : public exception {
public:
  explicit logic_error(const string& what_arg);
  ...
};
class domain_error : public logic_error {
public:
  explicit domain_error(const string& what_arg);
  ...
};
```

The logic_error family describes, as you might expect, typical logic errors:

```
domain_error
invalid_argument
length_error
out_of_bounds
```

Next, the runtime_error family describes errors that might show up during runtime but that could not easily be predicted and prevented:

```
range_error
overflow_error
underflow_error
```

klasa stdexcept

For example, the following code catches the out_of_bounds exception individually, treats the remaining logic_error family of exceptions as a group and treats exception objects, the runtime_error family of objects and any remaining exception types derived from exception collectively:

```
try {
  ...
}
catch (out_of_bounds & oe) // catch out_of_bounds error
{
  ...
}
catch (logic_error & oe) // catch remaining logic_error family
{
  ...
}
catch (exception & oe) // catch runtime_error, exception objects
{
  ...
}
```

If one of these library classes doesn't meet your needs, it makes sense to derive an exception class from logic_error or runtime_error so that you can fit your exceptions into the same general hierarchy.

klasa stdexcept

For example, the following code catches the out_of_bounds exception individually, treats the remaining logic_error family of exceptions as a group and treats exception objects, the runtime_error family of objects and any remaining exception types derived from exception collectively:

```
try {
  ...
}
catch (out_of_bounds & oe) // catch out_of_bounds error
{
  ...
}
catch (logic_error & oe) // catch remaining logic_error family
{
  ...
}
catch (exception & oe) // catch runtime_error, exception objects
{
  ...
}
```

If one of these library classes doesn't meet your needs, it makes sense to derive an exception class from logic_error or runtime_error so that you can fit your exceptions into the same general hierarchy.

wyjątek bad_alloc i instrukcja new

```
// newexcp.cpp -- the bad_alloc exception
#include <iostream>
#include <new>
#include <cstdlib> // for exit(), EXIT_FAILURE
using namespace std;
struct Big
{
  double stuff[20000];
};
int main()
{
  Big* pb;
  try {
    cout << "Trying to get a big block of memory:\n";
    pb = new Big[10000]; // 1,600,000,000 bytes
    cout << "Got past the new request:\n";
  }
  catch (bad_alloc & ba)
  {
    cout << "Caught the exception!\n";
    cout << ba.what() << endl;
    exit(EXIT_FAILURE);
  }
  cout << "Memory successfully allocated\n";
  pb[0].stuff[0] = 4;
  cout << pb[0].stuff[0] << endl;
  delete[] pb;
  return 0;
}
```

```
Big* pb;
pb = new (std::nothrow) Big[10000]; // 1,600,000,000 bytes
if (pb == 0)
{
  cout << "Could not allocate memory. Bye.\n";
  exit(EXIT_FAILURE);
}
```

Obsługa błędów – wyjątki

Podsumowanie :

- Zwracanie kodu błędu
- Mechanizm wyjątków
- Wyjątki w postaci obiektów
- Rozwijanie stosu (różnice między throw a return)
- Przekazanie wyjątku przez wartość i przez referencję
- Klasa exception

Programowanie generyczne, szablony

Programowanie generyczne, szablony

Plan zajęć :

- Szablony funkcji
- Przeciążanie szablonów
- Specjalizacje jawne
- Szablony klas

Programowanie generyczne, szablony

- Programowanie uogólnione (rodzajowe, generyczne, z ang. generic programming) – jeden z paradygmatów programowania.
- Programowanie uogólnione pozwala na pisanie kodu programu bez wcześniejszej znajomości typów danych, na których kod ten będzie pracował.
- Szablony umożliwiają programowanie uogólnione (ang. generic programming).
- **Szablon funkcji to ogólny opis** funkcji, czyli taki, **w którym funkcja jest opisana przez typy ogólne** – pod nie można później podstawić typy konkretne , jak *int* czy *double*.

Szablony funkcji

```
// funtemp.cpp -- użycie szablonu funkcji
#include <iostream>
// prototyp szablonu funkcji
template <typename T> // albo class T
void Swap(T& a, T& b);

int main()
{
  setlocale(LC_ALL, "pl_PL");

  using namespace std;
  int i = 10;
  int j = 20;
  cout << "i, j = " << i << ", " << j << ".\n";
  cout << "Użycie funkcji obsługującej typ int, "
    "wygenerowanej przez kompilator:\n";
  Swap(i, j);  // generuje void Swap(int &, int &)
  cout << "Teraz i, j = " << i << ", " << j << ".\n";

  double x = 24.5;
  double y = 81.7;
  cout << "x, y = " << x << ", " << y << ".\n";
  cout << "Użycie funkcji obsługującej typ double, "
    "wygenerowanej przez kompilator:\n";
  Swap(x, y);  // generuje void Swap(double &, double &)
  cout << "Teraz x, y = " << x << ", " << y << ".\n";
  // cin.get();
  return 0;
}

// definicja szablonu funkcji
template <typename T> // lub class T
void Swap(T& a, T& b)
{
  T temp;   // zmienna temp typu T
  temp = a;
  a = b;
  b = temp;
}
```

Przeciążanie szablonów

```
// twotemps.cpp -- użycie przeciążonych szablonów funkcji
#include <iostream>
template <typename T>     // szablon oryginalny
void Swap(T& a, T& b);

template <typename T>     // nowy szablon
void Swap(T* a, T* b, int n);

void Show(int a[]);
const int Lim = 8;
int main()
{
  setlocale(LC_ALL, "pl_PL");

  using namespace std;
  int i = 10, j = 20;
  cout << "i, j = " << i << ", " << j << ".\n";
  cout << "Użycie funkcji obsługującej typ int, "
    "wygenerowanej przez kompilator:\n";
  Swap(i, j);              // pasuje do szablonu oryginalnego
  cout << "Teraz i, j = " << i << ", " << j << ".\n";

  int d1[Lim] = { 0,7,0,4,1,7,7,6 };
  int d2[Lim] = { 0,7,2,0,1,9,6,9 };
  cout << "Tablice początkowo:\n";
  Show(d1);
  Show(d2);
  Swap(d1, d2, Lim);        // pasuje do nowego szablonu
  cout << "Tablice po zamianie:\n";
  Show(d1);
  Show(d2);
  // cin.get();
  return 0;
}
```

```
template <typename T>
void Swap(T& a, T& b)
{
  T temp;
  temp = a;
  a = b;
  b = temp;
}

template <typename T>
void Swap(T a[], T b[], int n)
{
  T temp;
  for (int i = 0; i < n; i++)
  {
    temp = a[i];
    a[i] = b[i];
    b[i] = temp;
  }
}

void Show(int a[])
{
  using namespace std;

  for (int i = 0; i < Lim; i++)
    cout << a[i] << "\t";

  cout << endl;
}
```

Specjalizacje jawne

```
#include <iostream>
template <typename T>
void Swap(T& a, T& b);

struct point
{
  char name[40];
  double x;
  double y;
};

// jawna specjalizacja
template <> void Swap<point>(point& j1, point& j2);
void Show(point& p);

int main()
{
  setlocale(LC_ALL, "pl_PL");

  using namespace std;
  cout.precision(2);
  cout.setf(ios::fixed, ios::floatfield);
  int i = 10, j = 20;
  cout << "i, j = " << i << ", " << j << ".\n";
  cout << "Użycie generowanej przez kompilator funkcji "
    "zamieniającej wartości int:\n";
  Swap(i, j);    // generuje void Swap(int &, int &)
  cout << "Teraz i, j = " << i << ", " << j << ".\n";

  point p1 = { "Początek wektora", 0.0, 0.0 };
  point p2 = { "Koniec wektora", 10.5, 25.7 };
  cout << "Przed zamianą struktur point:\n";
  Show(p1);
  Show(p2);
  Swap(p1, p2); // używa void Swap(point &, point &)
  cout << "Po zamianie struktur point:\n";
  Show(p1);
  Show(p2);
  // cin.get();
  return 0;
}
```

```
template <typename T>
void Swap(T& a, T& b)    // wersja ogólna
{
  T temp;
  temp = a;
  a = b;
  b = temp;
}

// zamienia tylko pola salary i floor struktury job

template <> void Swap<point>(point& p1, point& p2)  // specjalizacja
{
  double t;
  t = p1.x;
  p1.x = p2.x;
  p2.x = t;
  t = p1.y;
  p1.y = p2.y;
  p2.y = t;
}

void Show(point& p)
{
  using namespace std;
  cout << p.name << ", x: " << p.x
    << ", y: " << p.y << endl;
}
```

Programowanie generyczne, szablony

- Dla danej nazwy funkcji można mieć funkcję nieszablonową, szablon funkcji oraz jawną specjalizację szablonu funkcji, a także przeciążone wersje ich wszystkich.
- Prototyp i definicja jawnej specjalizacji muszą być poprzedzone zapisem *template*<>, powinny wymieniać nazwę specjalizowanego szablonu.
- Specjalizacja przykrywa zwykły szablon, a funkcja zwykła przykrywa je wszystkie.

Szablony klas

```
// stacktp.h -- szablon stosu
#ifndef STACKTP_H_
#define STACKTP_H_
template <class Type>
class Stack
{
private:
  enum { MAX = 10 }; // stała specyficzna dla klasy
  Type items[MAX]; // przechowuje elementy stosu
  int top; // indeks elementu ze szczytu stosu
public:
  Stack();
  bool isempty();
  bool isfull();
  bool push(const Type& item); // dodaje element do stosu
  bool pop(Type& item); // zdejmuje top ze stosu i umieszcza w item
};

template <class Type>
Stack<Type>::Stack()
{
  top = 0;
}

template <class Type>
bool Stack<Type>::isempty()
{
  return top == 0;
}

template <class Type>
bool Stack<Type>::isfull()
{
  return top == MAX;
}
```

```
template <class Type>
bool Stack<Type>::push(const Type& item)
{
  if (top < MAX)
  {
    items[top++] = item;
    return true;
  }
  else
    return false;
}

template <class Type>
bool Stack<Type>::pop(Type& item)
{
  if (top > 0)
  {
    item = items[--top];
    return true;
  }
  else
    return false;
}

#endif
```

Szablony klas

```
// stacktem.cpp -- test szablonu klasy do obsługi stosu
#include <iostream>
#include <string>
#include <cctype>
#include "stacktp.h"
using std::cin;
using std::cout;

int main()
{
  setlocale(LC_ALL, "pl_PL");

  Stack<std::string> st; // tworzy pusty stos
  char ch;
  std::string po;
  cout << "Wpisz D, aby dodać zamówienie,\n"
    << "P, aby przetworzyć zamówienie, i Z, aby zakończyć.\n";
```

```
while (cin >> ch && std::toupper(ch) != 'Z')
  {
    while (cin.get() != '\n')
      continue;
    if (!std::isalpha(ch))
    {
      cout << '\a';
      continue;
    }
    switch (ch)
    {
    case 'D':
    case 'd': cout << "Podaj nowy numer zamówienia: ";
      cin >> po;
      if (st.isfull())
        cout << "stos pełny\n";
      else
        st.push(po);
      break;
    case 'P':
    case 'p': if (st.isempty())
      cout << "Stos pusty\n";
            else {
      st.pop(po);
      cout << "Numer zamówienia " << po << " zdjęty\n";
      break;
    }
    }
    cout << "Wpisz D, aby dodać zamówienie,\n"
      << "P, aby przetworzyć zamówienie, i Z, aby zakończyć.\n";
  }
  cout << "Koniec\n";
  return 0;
}
```

Programowanie generyczne, szablony

Podsumowanie :

- Szablony funkcji
- Przeciążanie szablonów
- Specjalizacje jawne
- Szablony klas

KONIEC