Programowanie obiektowe

dr inż. Piotr Szwed
Katedra Informatyki Stosowanej
C2, pok. 403

e-mail: pszwed@agh.edu.pl

http://home.agh.edu.pl/~pszwed/

Aktualizacja:16.11.2019

2. Składowe klas, kompozycja i dziedziczenie

Składowe klas

Defincje klas

W języku C++ klasy można definiować używając słów kluczowych `class`, `struct` lub `union`.

Definicja klasy najczęściej obejmuje:

- **atrybuty** (pola)
- **metody** (funkcje składowe)
- **jeden** lub **kilka konstruktorów**
- **jeden destruktor**

Deklaracja klas z użyciem słów `struct` i `union` jest zapewniona dla zgodności z językiem C. Standardowo, dostęp do ich metod i pól jest publiczny.

W przypadku użycia słowa kluczowego `class`, standardowo dostęp jest prywatny.

Deklaracja klasy

```
class File
{
    FILE*fp;
public:
    File(); // standardowy konstruktor
    File(const char*name, const char*mode); //
    ~File(); // destruktor
    int open(const char*name, const char*mode);
    int close();
    int get();
    int put(int);
};
```

Implementacja metod

```
File::File(){fp = 0;}

File::File(const char*name, const char*mode){
    open(name,mode);
}
int File::close(){
    if(fp)fclose(fp);
    fp=0;
    return 1;
}

int File::open(const char*name,const char*mode)
{
    fp = fopen(name,mode);
    return fp!=0;
}
```

Użycie klasy

```
void f(){
    File file; // obiekt typu File

    File file2("plikwy.txt","wt");

    File *pfile = new File("plikwe.txt","rt"");
    for(;;){
        int c = pfile->get();
        if(c<0)break;
        file2.put(c);
    }
    delete pfile;
}
```

Deklaracja klasy

Składnia:

```
class identifier[base-class-specifier]
{
      member-list
}  ;
```

- Nazwa klasy staje się widoczna dla kompilatora bezpośrednio po przetworzeniu nagłówka klasy.
- Deklaracja klasy wprowadza nowy identyfikator do przestrzeni nazw. Deklaracje te są równocześnie definicjami typu w danej jednostce translacji (kompilowanym pliku źródłowym).

Pliki h i cpp

Definicje klas są zazwyczaj używane w większej liczbie jednostek translacji, stąd typową praktyką jest umieszczenie ich w plikach nagłówkowych (*.h). Implementacje metod klas umieszcza się w plikach *.cpp (odrębnych jednostkach translacji).

a.h

```
class A
{
public:
   A();
};
```

b.h

```
#include "a.h"
class B
{
public:
   B();
   A a;
};
```

Definicja klasy B wymaga znajomości definicji klasy A, stąd nagłówek a.h jest włączany do nagłówka b.h.

Pliki h i cpp

Kompilator kompiluje jednostki translacji - pliki cpp wraz z włączonymi nagłówkami.

a.cpp	b.cpp	main.cpp

`#include "a.h"`
`A::A(){}`
	`#include "b.h"`
	`B::B(){}`
		`//#include "a.h"`
		`#include "b.h"`
		`int main()`
		`{`
		`  A a;`
		`  B b;`
		`  return 0;`
		`}`

Pliki h i cpp

W danej jednostce translacji może pojawić się **dokładnie jedna** definicja klasy. Problemem jest, śledzenie zależności pomiędzy plikami nagłówkowymi.

Najczęściej stosowanym zabezpieczeniem jest użycie dyrektyw warunkowej kompilacji:

a.h

```
#if !defined _a_h_
#define _a_h_

class A
{
public:
   A();
};

#endif // _a_h_
```

Przykład

```
#include "a.h" // (1)
#include "b.h" // (2)
void main()
{
      A a;
      B b;
}
```

1. Klasa A zostanie zdefiniowana. Zdefiniowany zostanie symbol preprocesora _**a**_**h**_
2. Włączony zostanie nagłówek b.h. Przetwarzanie b.h pociągnie włączenie a.h. Ponieważ _**a**_**h**_ istnieje w słowniku symboli preprocesora, powtórna definicja klasy A jest pomijana.

Elementy składowe klas

Klasy mogą mieć następujące elementy składowe:

- funkcje składowe (metody)
- przechowywane dane (pola, atrybuty)
- klasy zagnieżdżone (wewnętrzne)
- wyliczenia (enum)
- pola bitowe
- deklaracje klas zaprzyjaźnionych (friend)
- wewnętrzne deklaracje typów

Elementy klas mogą być zadeklarowane z użyciem modyfikatorów: const i static.

Metody – przykład w C 1

```
// deklaracja struktury
typedef struct {double x,y;}StrComplex;

// nadanie wartości początkowych lub przypisanie
void init(StrComplex*pc, double _x, double _y)
{
    pc->x=_x ;pc->y=_y;
}

// oblicza moduł liczby zespolonej
double module(const StrComplex*pc)
{
    return sqrt(pc->x*pc->x + pc->y*pc->y);
}
```

Metody – przykład w C 2

```
// wypisuje wartość składowych oraz moduł
void dump(const StrComplex*pc)
{
    printf("[x=%g, y=%g,module=%g]”,
    pc->x,pc->y, module(pc)) ;
}

void f(){
    StrComplex c;
    // nadajemy wartość początkową
    init(&c,2.4,3.76) ;
    // wypisujemy informacje
    dump(&c) ;
}
```

Metody - przykład w C++ 1

```
class Complex
{
public:
    double x,y;
    Complex(double _x,double _y)
            :x(_x),y(_y){}
    double module()const{
            return sqrt(x*x+y*y );
    }
    void dump()const;
    void set(double _x,double _y){
            x=_x;
            y=_y;}
};
```

Metody - przykład w C++ 2

```
void Complex::dump()const
{
    printf("[x=%g, y=%g,module=%g]”,
    this->x, this->y, this->module()) ;
}

void g() {
    Complex c(2.4, 3.76);
    c.dump();
    Complex *pc = &c;
    pc->dump();
    Complex &rc = c;
    rc.set(2.0, 3.0);
    rc.dump();
}
```

Porównanie 1

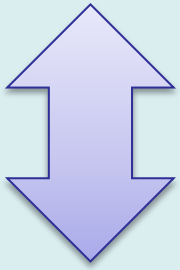
```
void init(StrComplex*pc, double _x, double _y)
{
    pc->x=_x ;pc->y=_y;
}
```

```
Complex(double _x,double _y):x(_x),y(_y){}

void set(double _x,double _y){x=_x ;y=_y ;}
```

Porównanie 2

```
double module(const StrComplex*pc)
{
    return sqrt(pc->x*pc->x + pc->y*pc->y);
}
```



```
double module()const{return sqrt(x*x+y*y ) ;}
```

Porównanie 3

```
void dump(const StrComplex*pc)
{
    printf("[x=%g, y=%g,module=%g]”,
    pc->x,pc->y, module(pc)) ;
}
```

```
void Complex::dump()const
{
    printf("[x=%g, y=%g,module=%g]”,
    this->x, this->y, this->module()) ;
}
```

Metody obiektu

1. W metodach należących do obiektu możemy bezpośrednio odwoływać się do jego danych. (Domyślnie odwołujemy się do tego obiektu, którego metoda jest wołana – `this`.)
2. W funkcjach składowych należących obiektu możemy wołać inne metody danego obiektu.
3. Wołając metody spoza obiektu wskazujemy obiekt, do którego wysyłamy komunikaty podając nazwę obiektu, wskaźnik lub referencję.
4. Funkcje, które nie modyfikują obiektu mogą być zadeklarowane jako `const`.

```
void dump(const StrComplex*pc);
void dump()const;
```

1. Funkcje składowe mogą być implementowane wewnątrz definicji klasy lub poza nią – `dump()`.

Wskaźnik this 1

- Wewnątrz niestatycznych metod obiektu można posługiwać się niemodyfikowalnym (`const`) wskaźnikiem this do obiektu danej klasy. Jest on domyślnym ukrytym argumentem każdej niestatycznej funkcji składowej.

```
CLASS * const this;
```

- Wewnątrz metod zadeklarowanych jako const (nie mających prawa modyfikować zawartości obiektu) wskaźnik this jest widoczny jako:

```
const CLASS * const this;
```

Wskaźnik this 2

```
class Light
{
    double voltage;
public:
    void on(){this->voltage = 230;}
    void off(){this->voltage = 0;}
    void brighten(){if(this->voltage<=220) this->voltage+=10;}
    void dim(){if(this->voltage>=10) this->voltage-=10;}
};
```

```
Light lt;
lt.on();
lt.dim();
lt.off();
```

```
Light*plt = new Light();
plt->on();
plt->dim();
plt->brighten();
delete plt;
```

Adres `&lt` lub `plt` są dostarczane do metod obiektu jako ich pierwszy (ukryty) argument

Wskaźnik this 3

Za pośrednictwem wskaźnika `this` można realizować dostęp do funkcji składowych i danych.

- W niektórych przypadkach pomaga to rozwiązać niejednoznaczności.
- Wskaźnika `this` używa się także często przy konieczności zwrócenia referencji do danego obiektu.

```
Complex& Complex ::set(double x,double y)
{
    this->x=x ;
    this->y=y ;
    return *this ;
}
```

Wskaźnik this 4

Może on służyć do ustalania asocjacji (powiązania) pomiędzy obiektami.

```
class Owner;
class Child
{
public:
    Owner*owner;
};
```

```
class Owner
{
    list<Child*> children;
public:
    void add (Child*child){
        child->owner=this;
        children.push_back(child);
    }
};
```

Funkcje inline 1

Funkcje inline, to funkcje, których wywołanie jest bezpośrednio zastępowane kodem funkcji. W przypadku bardzo krótkich funkcji ich użycie jest bardziej ekonomiczne, ponieważ znika dodatkowy narzut na wywołanie funkcji, powrót z wywołania oraz przesyłanie w obie strony danych poprzez stos. Wygenerowany kod może działać szybciej i być mniejszy.

```
class Int
{
    int value;
public:
    Int(int v):value(v){}
    int get()const{return value;} // inline
    void set(int v){value = v;} // inline
};
```

Funkcje inline 2

Funkcje zaimplementowane wewnątrz definicji klasy w miarę możliwości są tłumaczone jako funkcje inline. Alternatywnie, funkcje które są implementowane poza definicją klasy mogą być kompilowane jako funkcje inline po poprzedzeniu ich słowem kluczowym `inline` (traktowanym jako wskazówka dla kompilatora).

```
inline void Complex::dump()const
{
    printf("[x=%g, y=%g,module=%g]”,
    this->x, this->y, this->module()) ;
}
```

Kompilator ignoruje słowo kluczowe inline w przypadku, kiedy funkcja jest funkcją rekurencyjną lub może być wywoływana za pośrednictwem wskaźnika.

Składowe typu static

- W klasie można deklarować zarówno pola, jak i metody typu `static`. Traktuje się je jako elementy składowe klasy, a nie obiektu, stąd mogą być one dzielone przez wszystkie obiekty danej klasy, a także używane z zewnątrz.
- Metody statyczne nie mają dostępu do wskaźnika `this`, ponieważ w ich przypadku brak jest obiektu, który mógłby wskazywać. Stąd, w metodach statycznych można używać wyłącznie danych zadeklarowanych jako statyczne.

Składowe typu static

```
class A
{
public:
    A(){instanceCounter ++;}
    ~ A(){instanceCounter --;}
    static int getInstaceCounter()
    {return instanceCounter;}
    void dump()const;
private:
    static int instanceCounter;
};

void A::dump()const
{
    printf("A has %d instances", getInstaceCounter());
}
```

Możliwy dostęp do statycznego atrybutu
Brak dostępu do this

Statyczny atrybut

Składowe typu static

Pola statyczne klasy istnieją niezależnie od tego, czy istnieje jakikolwiek obiekt danej klasy. Z tego powodu pojawienie się statycznych pól w definicji klasy jest traktowane jak deklaracja `extern` , która wymaga odrębnej definicji, podczas której można także inicjować zmienne statyczne wartościami początkowymi.

```
int A::instanceCounter=0;
```

Składowe typu static

Metody statyczne mogą być wołane:

- z metody niestatycznej (poprzez bezpośrednie użycie nazwy)

```
void A::dump()const{
    printf("A has %d instances", getInstaceCounter());
}
```

- z zewnątrz za pośrednictwem obiektu

```
A a;
int n = a.getInstaceCounter();
```

- z zewnątrz poprzez podanie operatora zasięgu (*scope*)

```
printf("%d", A::getInstaceCounter());
```

Podobne reguły dotyczą zasad dostępu do statycznych atrybutów klasy.

Klasy zagnieżdżone (wewnętrzne)

- Klasy zagnieżdżone (*ang*. *nested, inner class*) są to klasy zadeklarowane wewnątrz innej klasy. Najczęściej stosuje się je jako pomocnicze struktury danych oraz tam gdzie nie chce się wprowadzać nowej nazwy do przestrzeni nazw kolidującej z istniejącymi nazwami.
- Deklaracja klasy zagnieżdżonej jest jedynie deklaracją typu

Klasy zagnieżdżone (wewnętrzne)

```
class Point {
    double x,y;
public:
    Point(double _x,  double _y);
};
```

```
class Circle
{
public:
    Circle(double _x, double _y, double r);
    class Point{
        double t[2] ;
    public:
        Point(double _x,  double _y);
    };
    Point center;
    double radius ;
};
```

Dwie klasy `Point` różniące się implementacjami:

- globalna
- wewnętrzna

Klasy zagnieżdżone (wewnętrzne)

```
// konstruktor klasy globalnej
Point::Point(double _x,  double _y){
    x=_x;
    y=_y;
}

// konstruktor klasy wewnętrznej
Circle::Point::Point(double _x,  double _y){
    t[0]=_x;
    t[1]=_y;
}

// konstruktor klasy zwewnętrznej
Circle::Circle(double _x, double _y, double r)
    :center(_x,_y),radius(r){}
```

Klasy zagnieżdżone (wewnętrzne)

- Używając klasy zagnieżdżonej w metodach klasy zewnętrznej możemy posługiwać się bezpośrednio nazwą klasy.
- Poza metodami klasy zewnętrznej musimy podawać pełną nazwę, postaci `Outer::Inner`,

```
Circle::Point pointInCircle;
```

- Dostęp do definicji klas zagnieżdżonych jest sterowany standardowymi modyfikatorami dostępu. Z zewnątrz nie możemy uzyskać dostępu do **definicji klasy** , jeżeli dostęp do niej jest ograniczony jako `private` lub `protected`.

Przykład

```
class Map
{
protected:
    class Pair{
        friend class Map;
        double x;
        double y;
    };
    Pair tab[1000];
    int cnt;
public:
    bool add(double x,double y){
        if(cnt==1000)return false;
        tab[cnt].x=x;
        tab[cnt].y=y;
        cnt++;
    }
    double get(double x){
        for(int i=0;i<cnt;i++){
            if(fabs(tab[i].x-x)<1e-5)return tab[i].y;
        }
        return -1e308;
    }
};
```

Klasa wewnętrzna

Typy wewnętrzne

Wewnątrz klas można deklarować zagnieżdżone typy za pośrednictwem konstrukcji `typedef`. Zasady dostępu do zagnieżdżonej definicji typu są analogiczne jak w przypadku klas.

```
class Tree
{
public:
    typedef Tree * PTREE;
    PTREE Left;
    PTREE Right;
    void *vData;
};

PTREE pTree;  // Error: not in class scope.
Tree::PTREE pTree;  // Ok.
```

Wyliczenia

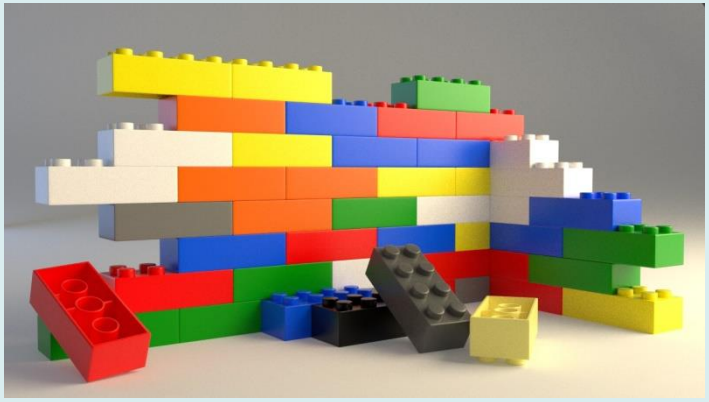
```
class HasState
{
    int state;
public:
    enum {good=0, bad, };
    HasState(){
        state = good;}
    int getState()const
    {return state;}
};
```

```
HasState hs;
int s = hs.getState();
if(s== HasState::good)     printf("good");
if(s== HasState::bad)  printf("bad");
```

Kompozycja i dziedziczenie

Kompozycja i dziedziczenie

- Techniki programowania obiektowego zyskały bardzo dużą popularność ze względu na możliwość łatwego wykorzystania istniejącego kodu (ang. *reuse*).
- Biblioteki obiektowe definiują zazwyczaj zbiór użytecznych komponentów, które następnie bez żadnych zmian oryginalnego kodu można włączać do aplikacji.



Kompozycja i dziedziczenie

Wykorzystanie istniejących komponentów może być realizowane dwiema drogami:

- przez **kompozycję** (agregację) czyli użycie istniejącego obiektu jako podobiektu (atrybutu) nowej klasy;

- przez **dziedziczenie**, czyli stworzenie nowej klasy obiektów rozszerzających funkcjonalność istniejącej klasy.

Kompozycja - komponent

```
class Temperature{
    double  value;
public:
    Temperature(double v = 0):value(v){}
    double getKelvin(){return value;}
    double  getCelsius(){return value-272.15;}
    double getFahrenheit(){return value * 9 / 5 - 459.67;}
    void setKelvin(double v){value = v;}
    void setCelsius(double v){value = v + 272.15;}
    void setFahrenheit(double v){value=(v+459.67)*5/9;}
};
```

- Przechowuje wartość temperatury w stopniach Kelvina
- Potrafi dokonać konwersji do stopni Celsiusza i Fahrenheita

Kompozycja

- Tworzymy klasę `Weather` (pogoda)
- Ma przechowywać:
  - Stan nieba (słońce, chmury, deszcz)
  - Temperaturę
  - Wilgotność
  - Prędkość wiatru

```
class Weather{
public:
    enum sky_state{sun,partly_cloudy,cloudy,rain,snow};

    enum sky_state sky;
    Temperature temp;
    double humidity;
    double windSpeed;
   //...
```

Kompozycja

- Dodajemy konstruktor
- Opcjonalnie: funkcje dostępu (settery i gettery)

```
//...
    Weather(double t, double h, double w, sky_state s):
        temp(t),humidity(h),windSpeed(w),sky(s){
    }
    void setKelvin();
    void setCelsius(double v);
    void setFahrenheit(double v);
    double getKelvin();
    double getCelsius();
    double getFahrenheit();
};
```

Kompozycja - delegacja

- `Weather` **deleguje** wykonanie metod do obiektu `temp`

```
double Weather::getKelvin(){
    return temp.getKelvin();
}

double Weather::getCelsius(){
    return temp.getCelsius();
}
double Weather::getFahrenheit(){
    return temp.getFahrenheit();
}
```

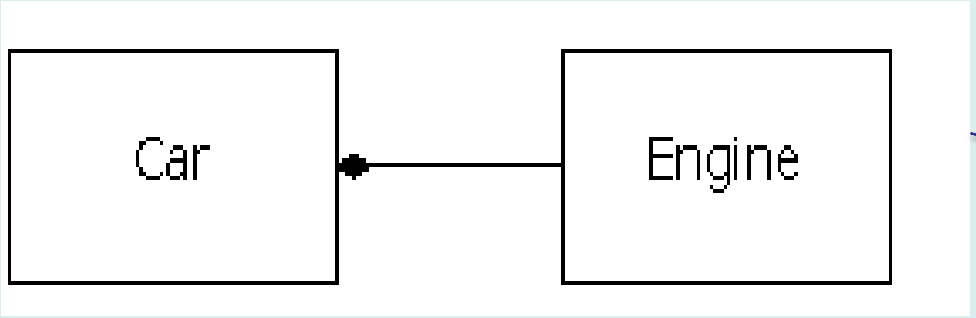
- Napisz settery…
- Jak skłonić kompilator aby utworzył funkcje inline

Kompozycja – analiza obiektowa

Aby podczas analizy sprawdzić, czy pomiędzy klasami zachodzi agregacja lub kompozycja stosujemy frazy języka naturalnego:

- ma, zawiera, obejmuje (agregacja)
- składa się (kompozycja)

Pogoda (dane opisujące pogodę) składają się z temperatury, wilgotności, prędkości wiatru i stanu nieba.



Reprezentacja graficzna

Dziedziczenie

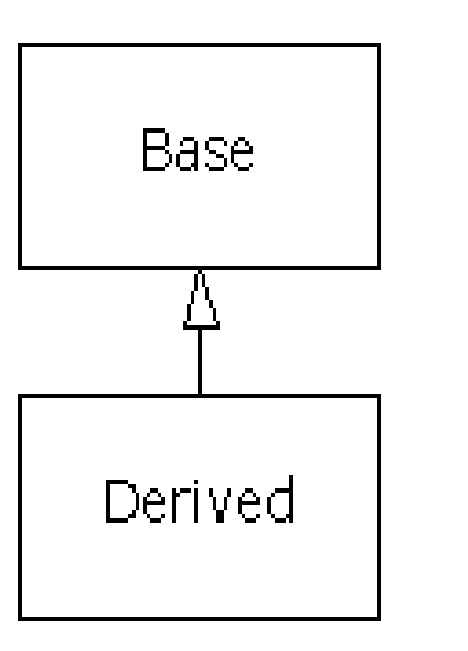
- Dziedziczenie umożliwia na stworzenie nowej defincji klasy (tzw. *klasy potomnej*) wykorzystującej istniejącą klasę (*klasę bazową*).
- Interfejs klasy bazowej jest w pełni zachowany. Obiekt klasy potomnej może przyjmować te same komunikaty – a więc należy również do klasy bazowej.
- Klasa potomna może dodawać nowe elementy do interfejsu, a także w odmienny sposób reagować na komunikaty zdefiniowane w interfejsie klasy bazowej.

Dziedziczenie

Często klasa bazowa nazywana jest *generalizacją* natomiast klasa potomna *specjalizacją*.

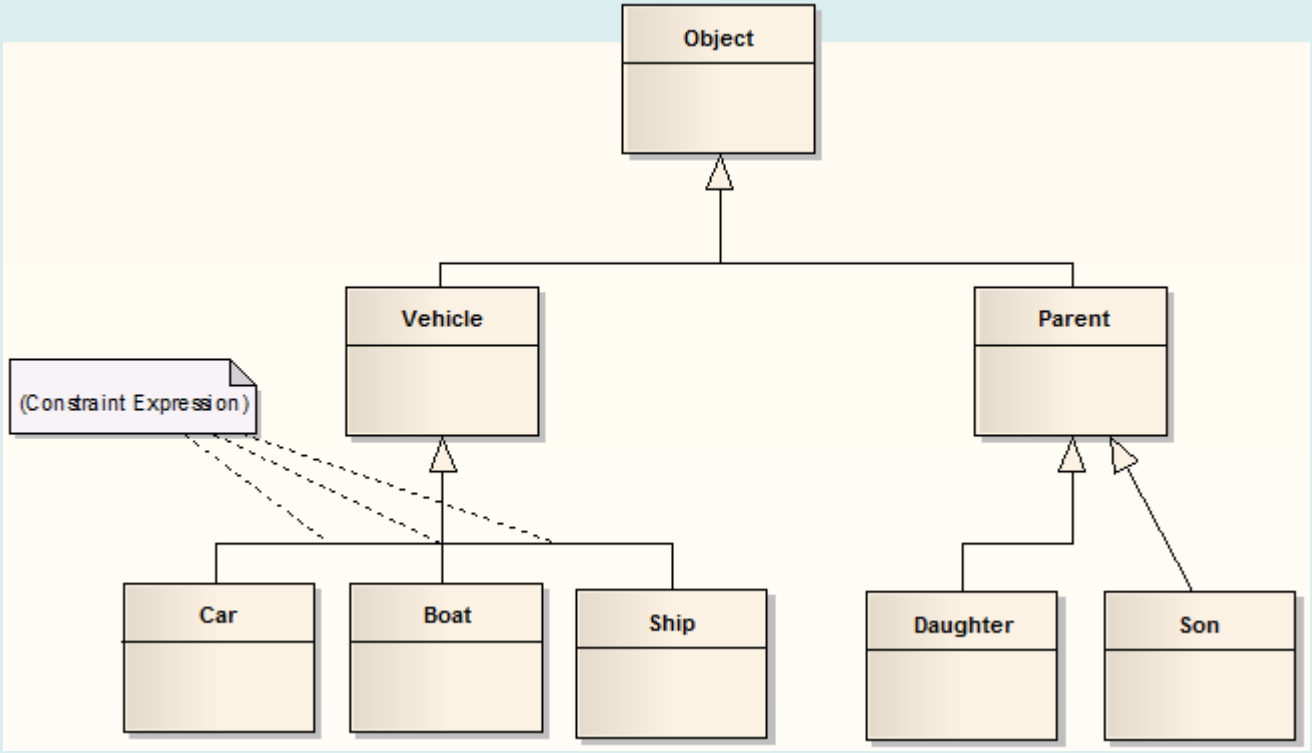
Decydując się na ustalenie, czy pomiędzy obiektami należy wprowadzić relację dziedziczenia posługujemy się frazą „*jest, jest rodzajem*”

- `B jest (jest rodzajem) A`
- `Okrąg jest Kształtem`
- `Trójkąt jest Kształtem`
- `SamochódOsobowy jest Pojazdem`



Reprezentacja graficzna

Dziedziczenie

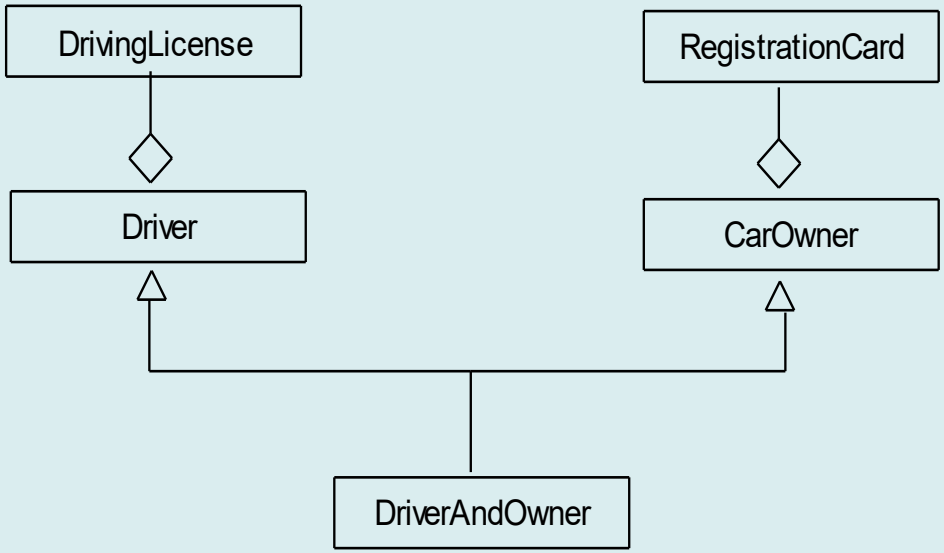


- Klasy połączone relacją dziedziczenia mogą tworzyć rozbudowane hierarchie
- W wielu językach mają one wspólny korzeń – klasę `Object`, stąd mówimy o hierarchiach obiektowych

Dziedziczenie wielokrotne (wielobazowe)

- Hierarchie obiektów są bardzo często rzutem opisu rzeczywistej dziedziny na konstrukcje języka programowania.
- Niejednokrotnie analizując dziedzinę, możemy podać obiekty, które należą do różnych klas, a równocześnie żadna z klas nie może zostać uznana za generalizację lub specjalizacją drugiej.

Dziedziczenie wielokrotne (wielobazowe)



- Charakterystyczną cechą klasy `Driver` jest posiadanie prawa jazdy (`DrivingLicense`). Charakterystyczną cechą posiadacza pojazdu `CarOwner` jest posiadania dowodu rejestracyjnego `RegistrationCard`.
- Bardzo często występują obiekty, które należą do obu klas równocześnie, a więc jak `DriverAndOwner` powinny dziedziczyć po dwóch klasach bazowych.

Dziedziczenie

Klasa `MeltingTemperature` (temperatura topnienia) rozszerza funkcjonalność temperatury – dodaje nazwę substancji.

```
class MeltingTemperature :public Temperature{
    string name;
public:
    MeltingTemperature(const char*n,double v):
       Temperature(v),name(n){
    }
    const char*getName(){return name.c_str();}
    // temperatura?
    bool setCelsius(double v){
        if(v<-272.15 || v>=2000) return false;
        setCelsius(v);
        return true;
    }
};
```

Dziedziczenie – przedefiniowanie metod

```
MeltingTemperature water("water",272.15);
cout<<water.getCelsius()<<endl;
water.setCelsius(-500);
cout<<water.getCelsius()<<endl;
water.Temperature::setCelsius(-500);
cout<<water.getCelsius()<<endl;
```

W obu przykładach wypisze

0
0
-500

```
MeltingTemperature water("water",272.15);
cout<<water.getCelsius()<<endl;

MeltingTemperature r1 = water;
r1.setCelsius(-500);
cout<<r1.getCelsius()<<endl;

Temperature r2 = water;
r2.setCelsius(-500);
cout<<r2.getCelsius()<<endl;
```

Wywołanie `setCelsius` w klasie bazowej

Składnia dziedziczenia

```
class derived : base-list {…}
base-list :
      base-specifier
      base-list , base-specifier
base-specifier :
      [virtual] [access-specifier] complete-class-name
access-specifier :
      private
      protected
      public
```

Przykłady

```
class X:public A {…} // jednobazowe
class Y:public A, public B {…} // wielobazowe
class Z: A, public B, protected C, virtual D
{…}
```

Specyfikacja dostępu

Słowa kluczowe `private`, `protected`, `public` definiują prawa dostępu w klasie pochodnej do elementów klasy bazowej. Brak słowa kluczowego (w przypadku deklaracji klasy) jest równoważny zastosowaniu dostępu `private`.

Dostęp w klasie bazowej	Sposób dziedziczenia	Dostęp w klasie pochodnej
public protected private	Public	public protected Brak dostępu
public protected private	Protected	protected protected Brak dostępu
public protected private	Private	Private private Brak dostępu

Przykład

Po zmianie definicji klasy `MeltingTemperature` na

```
class MeltingTemperature : private Temperature
{…}
```

za pośrednictwem klasy `MeltingTemperature` nie mamy dostępu z zewnątrz do żadnego komponentu (pola, metody, definicji) klasy `Temperature`.

`Temperature t(0);` `t.setCelsius(0);`	Poprawne
`MeltingTemperature mt;` `mt.Temperature::setCelsius(0);`	`Temperature::setCelsius` niedostępne jako metoda obiektu klasy `MeltingTemperature`
`Temperature&rt = mt;`	Konwersja typu jest potencjalnie możliwa, ale zabroniona przez tryb dziedziczenia

Przykład

```
class Light{
protected:
    double voltage;
public:
    Light(){voltage=0;}
    virtual void on(){voltage = 230;}
    virtual void off(){voltage=0;}
};
```

Klasa `Light` definiuje chroniony atrybut `voltage` i dwie publiczne metody `on()` oraz `off()`.

```
class DimmableLight:public Light{
    double level;
    void updateVoltage(){
        voltage=level*230;
        std::cout<<voltage<<std::endl;
    }
public:
    DimmableLight(){
        level=0;
        updateVoltage();}
    void dim(){
        if(level>=0.1)level -= 0.1;
        updateVoltage();}
    void brighten(){
        if(level<=0.9)level += 0.1;
        updateVoltage();}
    void on(){
        level=1.0;
        updateVoltage();}
    void off(){
        level=0;
        updateVoltage();}
};
```

- Klasa `DimmableLight` dziedziczy komponenty klasy `Light`.
- Dodano w niej prywatny atrybut `level` i prywatną metodę `updateVoltage();`
- Interfejs został rozszerzony o dwie dodatkowe metody `dim()` i `brighten()`.
- Dodatkowo, przedefiniowano metody `on()` i `off()`

Wywołanie

```
int main(){
    Light simpleLight;
    simpleLight.on();
    simpleLight.off();

    DimmableLight advancedLight;
    advancedLight.on();
    advancedLight.dim();
    advancedLight.dim();
    advancedLight.off();
}
```

Wynik:

```
0
230
207
184
0
```